CodeWarrior®

Pascal Compilers Reference

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 98/08/17 map

Metrowerks CodeWarrior copyright ©1993–1998 by Metrowerks Inc. and its licensors. All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 9801 Metric, Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	goto `Metrowerks`

Table of Contents

1 Introduction **9**

Introduction Overview 9
Read the Release Notes! 9
What's New 10
What's in the Rest of this Manual 10
Where To Go From Here 11
Programming for a Specific Platform 11
Conventions Used in This Manual 12

2 Pascal Settings **15**

Pascal Settings Overview 15
Pascal Language Settings 15
Activate Object Pascal. 17
ANS Conformance 17
Case Sensitive Identifiers 17
Copy Strings Using Length Byte 18
Copy Value Parameter 18
Debugging Aids 18
Don't Reuse Sets 19
Don't Reuse Strings. 19
Expand Method Tables 19
Inline Method Dispatching 19
IO Model 20
Optimize Class Hierarchy 20
Pointer Based Objects 20
Pool Sets 20
Pool Strings 21
Prefix File 21
Relax Pointer Compatibility 21
Uses Propagation. 22
Pascal Warnings Settings 23
Modified For-loop Indexes 24
Function Returns 24
Unused Variables. 25

Unused Arguments . . . 25
Undefined Routines . . . 26
Check String Parameter Sizes . . . 26
Gotos & Labels . . . 27
Branching Into With . . . 27
Branching Into For . . . 27
Branching Between Case Legs . . . 28
Branching Between If & Else . . . 28
Command Line Alternatives . . . 29

3 Using CodeWarrior Pascal 31
Using Pascal Overview . . . 31
Pascal File Name Extensions . . . 31
Using Precompiled Units . . . 32
Calling C Routines from Pascal . . . 32
Calling C++ Routines from Pascal . . . 34
Restrictions When Calling C++ Routines . . . 35
Matching C/C++ and Pascal Types . . . 36
Examples . . . 37

4 Directives and Predefined Symbols 39
Directives and Predefined Symbols Overview . . . 39
Compiler Directives . . . 39
Using Compiler Directives . . . 40
$ALIGN . . . 41
$D . . . 42
$DEFINE . . . 42
$DEFINEC . . . 42
$ELIF . . . 44
$ELIFC . . . 44
$ELSE . . . 44
$ELSEC . . . 45
$ENDIF . . . 45
$ENDC . . . 45
$ERROR . . . 46
$ERRORC . . . 46

$EXPORT 46
$I 47
$IF 47
$IFC 47
$IFDEF 48
$IFOPT 48
$IFNDEF 49
$IFNOTC 49
$INCLUDE 49
$J 50
$LIBEXPORT 50
$MAIN 51
$MACRO 51
$MC68020 51
$MC68881 52
$NR 52
$OV 53
$PARAMSTR 53
$PARAM 53
$POP 54
$PRAGMA 54
$PRAGMAC 54
$PROF 55
$PUSH 55
$R 55
$S 56
$SETC 58
$SC 58
$U 58
$UNDEF 59
$UNDEFC 59
$UNDEFINED 59
$UNUSED 60
$W 60
$Z 60

Pragmas 60
Using Pragmas 62
Pragma Syntax 63
a6frames 63
align 64
ANS_conformance 65
casesensitive 65
chkstrparam 66
code68020 66
code68881 66
codelisting 67
codesize 68
codeviewinfo 68
copyparam 69
d0_pointers 69
deadcode 69
deadstore 69
debuglisting 70
disable_register 70
dont_reuse_sets 70
dont_reuse_strings 70
expandtables 71
export 71
external 72
externalvar 72
far_code 72
fardata 72
farmtables 73
farstrings 73
force_active 74
force_public_data 74
fourbyteints 74
fp_contract 75
global_optimizer 75
IEEEdoubles 76

import 76
inlinedispatching 76
internal 77
intrinsic 77
IOmodel 77
lib_export 78
lifetime 78
logopt 78
loopinvariant 78
macsbug 79
makemap 79
mark 79
mpwc 80
near_code 80
noreturn 80
objcheck 80
objectPascal 81
oldstyle_symbols 81
optimization_level 82
optimizeclasses 82
optimize_for_size 83
optimize_for_space 83
optimize_for_speed 84
overflow 84
pack 84
pcrelstrings 85
peephole 86
poolsets 86
poolstrings 86
pop 87
precompile_target 87
profile 87
propagation 88
ptrbasedobject 88
push 88

rangecheck 89
readonlystrings 89
regcoloring 89
relaxedptrcheck 90
scheduling 90
shortcircuit 91
sideeffects 91
smart_code 92
strengthreduction 92
strsize 93
subexpression 93
sym 93
syminfo 94
toc_data 94
traceback 94
trapnil 95
trapnoelse 95
usespropagation 95
warn_checkgoto 96
warn_foridx 96
warn_gotocase 96
warn_gotofor 96
warn_gotoif 97
warn_gotowith 97
warn_illpragma 97
warn_noreturn 97
warn_undefroutine 98
warn_unusedarg 98
warn_unusedvar 98
Checking Options 98
Predefined Symbols 105

Index **109**

1 Introduction

This manual contains the information you need to successfully use the CodeWarrior Pascal compilers to target different platforms. For a complete language specification of CodeWarrior Pascal, see the *Pascal Language Reference*.

Introduction Overview

Welcome to the *Pascal Compilers Reference*. This manual explains details about using CodeWarrior Pascal to develop software.

This compiler reference is intended to be used in tandem with the *Pascal Language Reference*. The language reference contains a specification of the CodeWarrior Pascal programming language. You will find that some information—a very small amount—is duplicated between the two manuals, but it is important to understand that this manual is not a substitute for the Language Reference.

This chapter introduces the CodeWarrior Pascal compiler. The other topics in this chapter are:

- Read the Release Notes!
- What's New
- What's in the Rest of this Manual
- Where To Go From Here
- Programming for a Specific Platform
- Conventions Used in This Manual

Read the Release Notes!

By now, you probably have read the CodeWarrior release notes. If you haven't, please do so before using the CodeWarrior Pascal com-

piler. They contain important information about new features, bug fixes, incompatibilities, and late-breaking changes. You can find them in the Release Notes folder.

What's New

New and changed topics in this manual are:

- "Pascal File Name Extensions" on page 31—describes how to make sure files are recognized as Pascal source code files
- "Checking Options" on page 98—expanded to cover all more values for checking options

What's in the Rest of this Manual

Each chapter begins with an overview section that introduces the chapter. The table below outlines what's in the other chapters in the Pascal Compiler Guide.

Table 1.1 What's in this reference

This chapter…	Describes
Introduction Overview	This is the chapter you're reading now.
Pascal Settings Overview	The Pascal-related options in the settings panels of the CodeWarrior integrated development environment (IDE).
Using Pascal Overview	Information about the CodeWarrior implementation of Pascal, including how to use precompiled units and using C and C++ object code with Pascal.
Directives and Predefined Symbols Overview	A reference for all the compiler directives, C-style pragmas and predefined symbols the CodeWarrior Pascal compiler is designed to recognize and act on.

Where To Go From Here

If you have not read the Release Notes, please do so now. You can find them on your CodeWarrior CD, in the Release Notes folder.

If you are new to CodeWarrior:

- See the *CodeWarrior QuickStart Guide* for an overview of CodeWarrior, and pointers to the references available to you. *Quick Start* is packaged with your CDs, whether or not you purchased the printed documentation
- For more detailed information about the CodeWarrior integrated development environment (IDE), consult the *IDE User Guide.*

NOTE: You will find all the manuals mentioned in this section in the CodeWarrior Documentation folder on your CodeWarrior CD-ROM or your computer's hard drive.

Programming for a Specific Platform

What processor or operating system will you be developing software for with CodeWarrior Pascal? To learn about using CodeWarrior to develop software for a specific operating system or processor, refer to its *Targeting* manual. For example, to learn how to use CodeWarrior to create an application for Mac OS, read *Targeting Mac OS*.

A *Targeting* manual covers such topics as the tools you use to develop software for a particular platform, how to create and configure projects, a tutorial, special compiler code generation and debugging issues, and other important topics that apply to a platform.

Conventions Used in This Manual

This manual, like the rest of the CodeWarrior documentation set, follows a number of typographic conventions and conventions having to do with excerpted source code.

Typographic Conventions

This manual uses typographic conventions to distinguish between types of words and symbols:

- Ordinary text is in plain letters, the kind you are reading now.
- Special technical terms are printed *this way* when they are first defined. After that, they are treated as ordinary text.
- Elements of the Pascal language (or any other programming language) are printed `this way`. This helps you avoid confusing them with ordinary text.
- Artificial terms, which have meaning only in this book, are printed *this way*. These terms are used primarily to indicate parts of programming syntax that you replace with your own symbols.
- Menu commands, dialog box items, and other controls are printed **this way** when they are first mentioned. After that, they are treated as ordinary text.

Source Code Conventions

The following conventions for Pascal source code are used in this manual:

- Reserved words are printed in `ALL CAPITALS`.
- The names of procedures and functions are printed in `Initial Capitals`.

- The names of data types, constants, and variables are printed in `lowercase`.
- A sequence of two dots (..) indicates a range between two values. For example, 0 .. 127 means "0 through 127."

2

Pascal Settings

This chapter describe the options you can set in the IDE (Integrated Development Environment) to control how CodeWarrior Pascal translates source code and generates object code.

Pascal Settings Overview

The CodeWarrior IDE associates a number of important parameters with every programming project. You display and modify these settings by selecting ***Target*** **Settings** on the **Edit** menu, where *Target* is the name of a target in a project, then selecting a category of settings to modify using the list on the left-hand side of the **Target Settings** dialog box.

This chapter describes the settings for **Pascal Language** in the **Language Settings** branch of the navigator tree, including the options relating to common mistakes for which the compiler normally issues warnings.

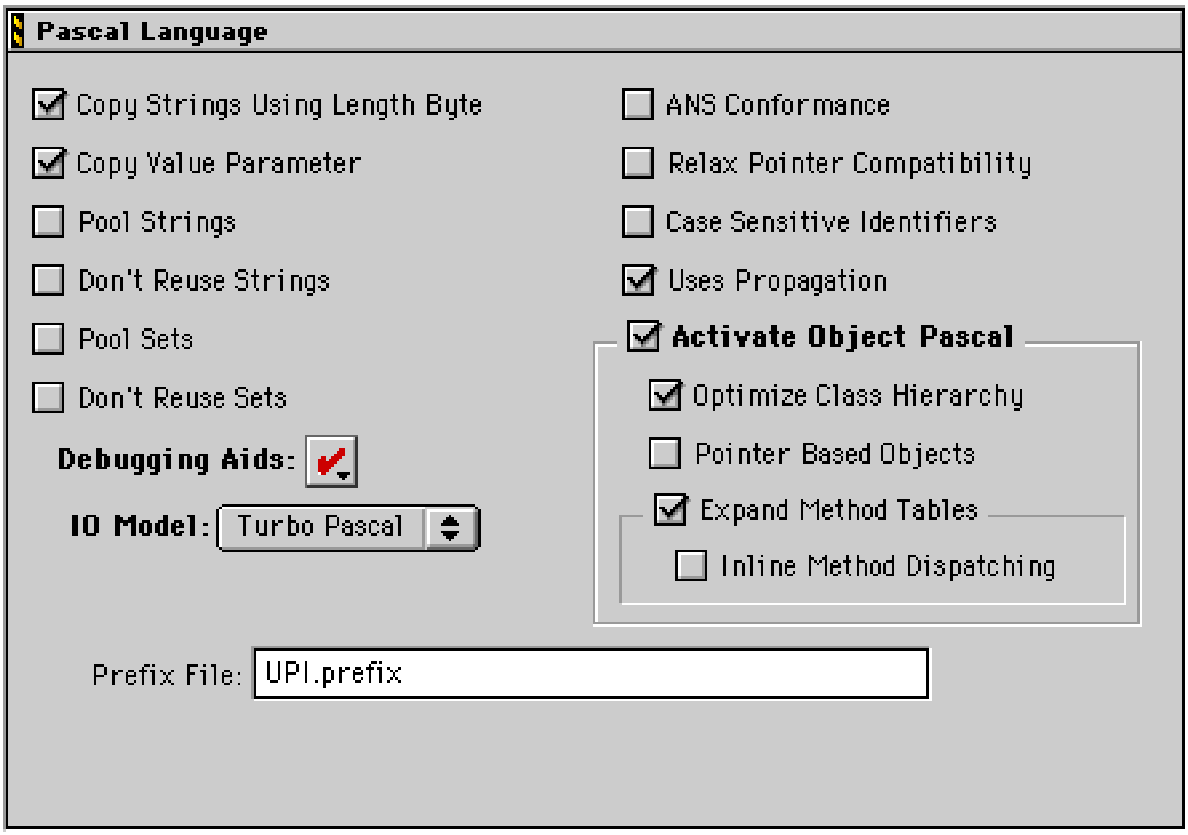
The rest of this chapter covers the following topics:

- Pascal Language Settings
- Pascal Warnings Settings
- Command Line Alternatives

Pascal Language Settings

The **Pascal Language** settings control how the CodeWarrior Pascal compiler translates your source code. For information about the related **Pascal Warnings** settings panel, which deals with common programming pitfalls that the compiler will warn about, see "Pascal Warnings Settings" on page 23.

Figure 2.1 The Pascal Language Settings panel



The rest of this section discusses the Pascal Language options listed below. To make them easier to locate quickly, they are described in alphabetical order rather than the order in which they appear in the Pascal Language settings panel.

- Activate Object Pascal
- ANS Conformance
- Case Sensitive Identifiers
- Copy Strings Using Length Byte
- Copy Value Parameter
- Debugging Aids
- Don’t Reuse Sets
- Don’t Reuse Strings

- Expand Method Tables
- Inline Method Dispatching
- IO Model
- Optimize Class Hierarchy
- Pointer Based Objects
- Pool Sets
- Pool Strings
- Prefix File
- Relax Pointer Compatibility
- Uses Propagation

Activate Object Pascal

Makes Object Pascal language extensions available. When this option is on, the compiler recognizes Object Pascal source code. When this option is off, the compiler will only recognize regular Pascal source code.

ANS Conformance

Checks source code for ANS Pascal conformance. When this option is on, the compiler only recognizes ANS Pascal source code. When this option is off, the compiler compiles non-standard Pascal source code. See *Pascal Language Reference* for information on CodeWarrior Pascal's extensions and exceptions to ANS Pascal.

Case Sensitive Identifiers

Distinguishes between uppercase and lowercase characters in user-defined identifiers.

For example, when this option is on, the variables "`Mynumber`" and "`MyNumber`" are treated as two separate variables. When the option is off, the compiler treats "`Mynumber`" and "`MyNumber`" as if they were the same variable.

Copy Strings Using Length Byte

Uses the actual length of the string stored in its length byte when manipulating the string. When this option is off, the compiler uses the declared length of the string.

If you initialize strings before using them, turn this option on to copy strings faster. If you don't, turn this option off.

Copy Value Parameter

Generates object code to copy value parameters for routine calls. Turning off this option prevents unnecessary copies of structured type value parameters, allowing the compiler to generate faster code to pass and access the parameters.

More specifically, when this option is off, the compiler searches through each value structured parameter. If a parameter has not been modified, and cannot be aliased to another parameter, then the parameter is accessed more efficiently.

Debugging Aids

This pop-up menu has options to control the generation of extra instructions for checking for possible problems during runtime.

Activate Range Checking

When selected, generates extra instructions to verify that array indices, subrange values, set values, and string values are not out of bounds.

Activate Nil Checking

When selected, generates extra instructions to verify that nil pointers aren’t dereferenced.

Activate Overflow Checking

When selected, generates extra instructions to check for arithmetic overflows.

Trap Unmatched Case

When selected, generates extra instructions to make sure that a value has a match in a `case` statement without an `otherwise` clause.

Don't Reuse Sets

Stores each set literal separately. If this option is off, the compiler stores only one copy of identical set literals. This option helps you save memory if your program contains lots of identical set literals.

Don't Reuse Strings

Stores each string literal separately. If this option is off, the compiler stores only one copy of identical string literals. This option helps you save memory if your program contains lots of identical string literals.

Expand Method Tables

Fills out object method tables to explicitly reference all object methods, including parents. This expansion uses more global memory, but improves method calling through a one-level table lookup instead of going through an object's ancestor hierarchy. This expansion builds method tables similar to C++ vtables (although they aren't compatible).

Inline Method Dispatching

Calls object methods by computing the method table lookup directly.

Combined with method table expansion, this option inlines the table lookup used for method calling.

IO Model

Chooses the set of input and output routines and behaviors you are using in your project (you must use one of these three):

- If your code follows ANS Pascal standards for input/output, select **ANS Pascal.**
- If your code follows Apple's MPW Pascal standards for input and output, select **MPW Pascal.**
- If your code follows the Turbo Pascal Input/Output model, select **Turbo Pascal.**

Optimize Class Hierarchy

Strips unused class descriptors and related methods that are not referenced in the final program, select the **Optimize Class Hierarchy** option in the Pascal Language preference panel.

Object code, including libraries and compiled units, compiled and linked with CW8 and earlier releases of CodeWarrior Pascal are not compatible with the new class hierarchy optimization.

Pointer Based Objects

Uses pointer based objects. To use handle based objects, deselect Pointer Based Objects.

Objects allocated with this option on are not compatible with handle-based objects. A version of `ObjIntf.p` supports both pointer and handle based objects.

Pool Sets

Collects all set constants into a single data object so your program needs one TOC entry for all of them. If this option is off, the compiler creates a unique data object and TOC entry for each set constant. Turning this option on decreases the number of TOC entries in your program.

This option is especially useful if your program is large and has many set constants, or uses the CodeWarrior Profiler.

Pool Strings

Collects all string constants into a single data object so your program needs one TOC entry for all of them. If this option is off, the compiler creates a unique data object and TOC entry for each string constant. Turning this option on decreases the number of TOC entries in your program.

This option is especially useful if your program is large and has many string constants or uses the CodeWarrior Profiler.

Prefix File

Reads and compiles a file before each source code file that the compiler translates. The file named in this text field is effectively inserted at the beginning of each source code file that the compiler processes, making any items declared in the prefix file the last scope searched before the "built-in" declarations. The compiler uses the current target's access paths to find the file named in this field.

Relax Pointer Compatibility

Allows pointer type variables to be compatible if they point to the same base type. With this option off, pointer type variables are compatible only if they are declared with the same type.

Listing 2.1 Relaxed pointer compatibility example

```
PROGRAM foo;

TYPE
  intptr = ^INTEGER;

VAR
  a,b : intPtr;
  c,d : ^INTEGER;
```

```
BEGIN
  (* a and b are always compatible *)
  b := a;

  (* c and d are always compatible *)
  c := d;

  (* d and a are compatible only with
    Relaxed Pointer Compatibility on *)
  d := a;
END.
```

This option is used to compile the Mac OS Universal Interfaces since they contain forward pointer declarations and type aliases.

Listing 2.2 Forward pointer declarations and type aliases

```
TYPE
  foo = record
    a : fooptr; { forward pointer declaration }
    (* ... *)
  end;

  { type alias, the compiler can't link foo.a to barptr }
  fooptr = barptr;
(* ... *)
```

Uses Propagation

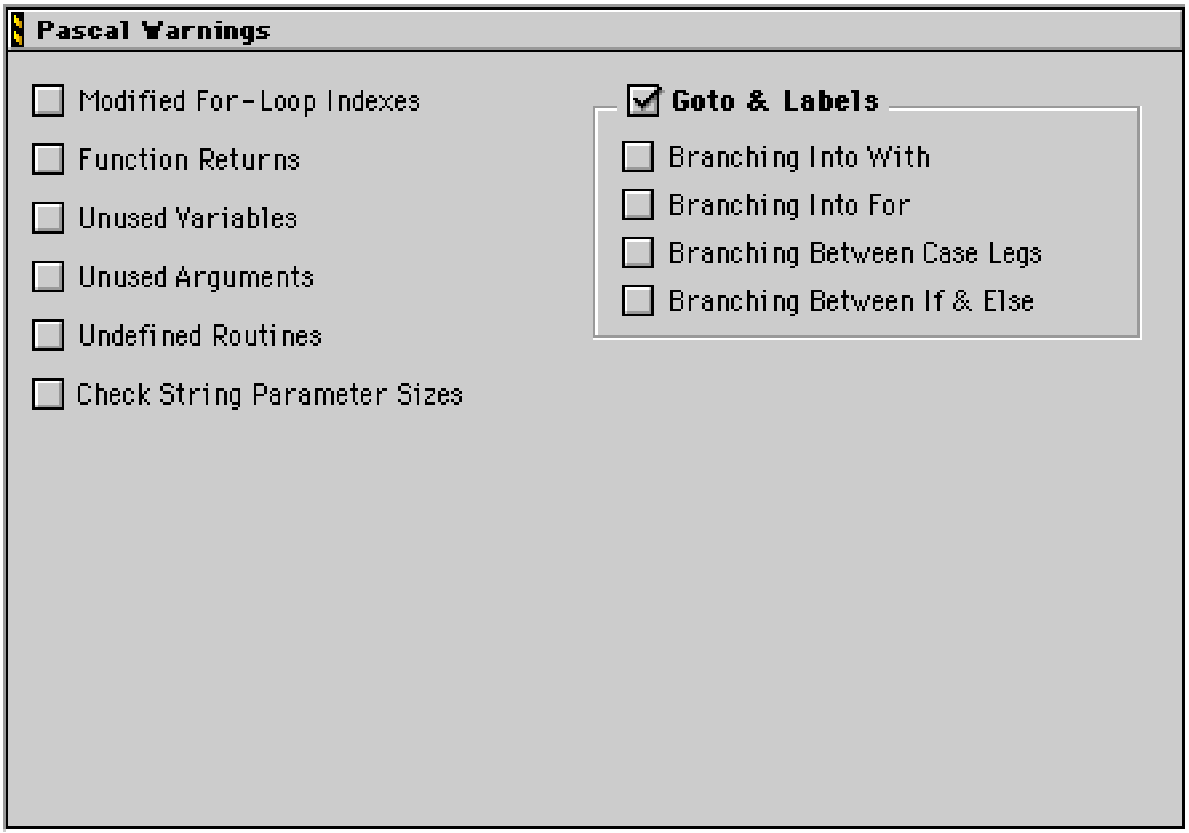
Extend the scope of unit objects among imported objects.

For more information, see "usespropagation" on page 95.

Pascal Warnings Settings

The CodeWarrior compiler has the advanced ability to warn you when you make some common Pascal programming mistakes. To turn on or off the compiler warning for each of these situations, choose **Target Settings** from the **Edit** menu, then select the **Pascal Warnings** settings panel.

Figure 2.2 The Pascal Warnings Settings panel



The rest of this section covers the following topics:

- Modified For-loop Indexes
- Function Returns
- Unused Variables
- Unused Arguments

- Undefined Routines
- Check String Parameter Sizes
- Gotos & Labels
- Branching Into With
- Branching Into For
- Branching Between Case Legs
- Branching Between If & Else

Modified For-loop Indexes

Generates a warning whenever a FOR loop index variable is assigned a new value either through an assignment statement or by passing that variable as an actual parameter to a VAR-parameter (Listing 2.3).

Listing 2.3 Example of Modified FOR-loop Indexes warning

```
PROCEDURE ForLoop;

VAR
  i : INTEGER;

BEGIN
  FOR i := 0 TO 10 DO
    i := i + 1; (* Generates a warning *)
END;
```

Function Returns

Generates a warning whenever a function's identifier doesn't get assigned at least one return value in the function's body.

Listing 2.4 Example of Function returns warning (Pascal)

```
FUNCTION PlusTwo(i : INTEGER) : INTEGER;

BEGIN
```

```
  i := i + 2;
(*Generates a warning because PlusTwo isn't
  assigned a value.
*)
END;
```

Unused Variables

Generates a warning whenever a local variable is declared but not used within a routine (Listing 2.5).

Listing 2.5 Example of Unused Variables warning

```
FUNCTION Triangle(w, h : REAL) : REAL;
VAR
(* This generates a warning;
   declared, but never used *)
  temp : REAL;

BEGIN
  Triangle := 0.5 * w * h;
END;
```

Unused Arguments

Generates a warning whenever an routine argument is not used within a routine (Listing 2.6).

Listing 2.6 Example of Unused Arguments warning (Pascal)

```
FUNCTION CubeVolume(w, h, d : REAL) : REAL;

BEGIN
(* Generates a warning
   because h and d are not used. *)
  CubeVolume := w * w * w;
END;
```

Undefined Routines

Generates a warning whenever a routine is declared in a unit `INTERFACE` part or has a `FORWARD` declaration but no definition (Listing 2.7).

Listing 2.7 Example of Undefined Routines warning (Pascal)

```
(* Generates a warning because second()
  is never defined *)
PROCEDURE second() ; FORWARD;

PROCEDURE first(VAR a : INTEGER);
BEGIN
  second();
  a := a + 10;
END;
```

Check String Parameter Sizes

Verifies declared string sizes and generates a warning for cases where memory mangling is likely.

This warning is useful because string type compatibility doesn't take length into account. This can produce interesting bugs when you pass by reference a shorter string to a longer one.

Listing 2.8 Example of Check String Parameter Sizes warning

```
(* Generates a warning because the string passed is shorter than
the referenced string *)
program testtext;

type str10 = string[10];

procedure x(var s:string); begin writeln(s); end;

var s : str10;
begin
```

```
    s := 'small str';
    x(s);
end.
```

Gotos & Labels

Allows a series of branching checks. These branching checks are:

- Branching Into With
- Branching Into For
- Branching Between Case Legs
- Branching Between If & Else

Branching Into With

Generates a warning whenever a label defined in a `WITH` statement gets referenced by a `GOTO` outside the `WITH` statement.

Listing 2.9 Example of Branching Into With warning (Pascal)

```
BEGIN
  GOTO 1234; (* Generates a warning *)
  WITH employee DO
  BEGIN
    name := 'Charles';
    age := 32;
    salary := 0;
1234:salary := salary + 30000;
  END; (* WITH employee *)
END.
```

Branching Into For

Generates a warning whenever a label defined in a `FOR` statement gets referenced by a `GOTO` outside the `FOR` statement (Listing 2.10).

Listing 2.10 Example of Branching Into For warning (Pascal)

```
BEGIN
  GOTO 1234; (* Generates a warning *)
  FOR i := 0 TO 20 DO
  BEGIN
1234: a := a + i;
  END;
END.
```

Branching Between Case Legs

Generates a warning whenever a label defined in one leg of a `CASE` statement gets referenced from another leg (Listing 2.11).

Listing 2.11 Example of Branching Between Case Legs warning (Pascal)

```
BEGIN
  CASE dayOfWeek of
    Sunday : StayHome();
    Monday : GOTO 1234; (* Generates warning *)
    Tuesday :
1234:       GotoWork();
  END; (* CASE dayOfWeek *)
END.
```

Branching Between If & Else

Generates a warning whenever a label defined in the `IF` part of an `IF`… `THEN`… `ELSE` statement is referenced from the `ELSE` part and vice-versa (Listing 2.12).

Listing 2.12 Example of Branching Between If and Else warning

```
BEGIN
  IF moonPhase = Full THEN
  BEGIN
    barkAtMoon := TRUE;
```

```
    GOTO 1234;
  END
  ELSE
1234:barkAtMoon := FALSE;
END.
```

Command Line Alternatives

Some versions of the CodeWarrior Pascal compilers are available as command-line programs, rather than being integrated into the IDE.

For detailed information on this topic, consult the CodeWarrior *Command-Line Tools Guide.*

3

Using CodeWarrior Pascal

This chapter describe covers various topics on using CodeWarrior Pascal, including taking advantage of precompiled units and using C and C++ routines with CodeWarrior Pascal.

Using Pascal Overview

This chapter covers topics on using CodeWarrior Pascal. This section covers these topics:

- Pascal File Name Extensions
- Using Precompiled Units
- Calling C Routines from Pascal
- Calling C++ Routines from Pascal
- Matching C/C++ and Pascal Types

Pascal File Name Extensions

By default, CodeWarrior IDE projects created using the **New Project** command are set up to recognized these file name extensions for the Pascal source code for programs and units: "`.p`" and "`.pas`". Add one of these extensions to your Pascal source code files to tell the IDE to compile these files using CodeWarrior Pascal.

Precompiled units for Pascal end with "`.ppu`". See "Using Precompiled Units" on page 32 for more information on precompiled units.

You may change, remove, or add to these file name extensions using your project's **File Mappings** settings panel. See the *IDE User Guide* for more information on using this settings panel.

Using Precompiled Units

To reduce the processing time needed to compile and re-compile commonly used units, CodeWarrior Pascal allows you to precompile units. A precompiled unit source file, a "`.ppu`" file, is just a regular unit that the Pascal compiler "pre-digests" into a precompiled unit, making compile times faster when the precompiled unit is referred to by other source code files. A precompiled unit file can be used as a prefix file. For related information, see "Prefix File" on page 21.

When using a precompiled unit as a prefix file, you need to make sure that you don't specify in the uses clause a unit that's used by the precompiled unit. Doing so will cause type mismatch errors because you're going to get two declarations for the same identifier.

You can specify a target name for a precompiled unit using

`{$pragma precompile_target` *filename* `}`

Note that the filename used must follow the syntax for pascal identifiers. For more information on this pragma see "precompile_target" on page 87.

Calling C Routines from Pascal

When calling a C function from Pascal, make sure the C function's object code is available to the Pascal program. The C function can be in a library or in a source code file in the Pascal project. Make sure the project also has any other libraries and source code files that the C function refers to.

To call a C function from Pascal:

1. **In the C source code, declare and define the C function.**

   If the C function has already been compiled into a library, then this step is unnecessary.

   Declare the C function, using a ANSI C-style prototype. Declaring a C function makes it available for external use, namely to be called by your Pascal source code.

   For example:

   ```
   short Sum(short i, short j);
   ```

   After the function's declaration, define it.

   For example:

   ```
   short Sum(short i, short j)
   {
     return i + j;
   }
   ```

2. **In the Pascal source code, declare the C function.**

   Before calling the C function in your source code, you must declare it. Use "`C; EXTERNAL`" to specify that the function uses C calling conventions and that it is external to the Pascal source code.

   For example, to declare the example function from step 1.:

   ```
   FUNCTION Sum(i, j:INTEGER):INTEGER; C; EXTERNAL;
   ```

   Note that the types of parameters and the return type in this Pascal declaration matches the C source code.

3. **In the Pascal source code, call the C function.**

   After the C function has been declared in your Pascal source code, you can call it like any other Pascal routine.

   For example:

   ```
   BEGIN
     total := Sum(subtotal1, subtotal2);
     (* ... *)
   END
   ```

   To determine how to match C types to Pascal types, follow the guidelines shown in "Matching C/C++ and Pascal Types" on page 36.

Calling C++ Routines from Pascal

When calling a C++ routine from Pascal, make sure the C++ routine's object code is available to the Pascal program. The C++ can be in a library or in a source code file in the Pascal project. Make sure the project also has any other libraries and source code files that the C function refers to.

To call a C++ function or object method from Pascal:

1. **In the C++ source code, create a C "glue" function.**

   Pascal can only refer to C functions, not C++ functions. To get around this limitation, use a C function that calls the C++ routine you want to call.

   In C++ source code, declare a C "glue" function, using a ANSI C-style prototype, with the `extern "C"` linkage specification.

   For example:

```
extern "C"
{
  short SumInCPP(short a, short b);
}
```

   After the glue function's declaration, define it and the C++ code that it calls.

   For example:

```
class SumClass
{
   short total;
   public:
   fee(short a, short b) :total(a+b) {}
   short sum() { return total; }
};

short SumInCPP(short a, short b)
{
  SumClass sumObj(a,b);
  return sumObj.sum();
}
```

2. **In the Pascal source code, declare the glue function.**

   Before calling the glue function in your source code, you must declare it. Use "`C; EXTERNAL`" to specify that the function uses C calling conventions and that it is external to the Pascal source code.

   For example, to declare the example function from step 1.:

   ```
   FUNCTION SumInCPP(i, j : INTEGER) : INTEGER;
   C; EXTERNAL;
   ```

   Note that the types of parameters and the return type in this Pascal declaration matches the glue function's C source code.

3. **In the Pascal source code, call the glue routine.**

   After the C function has been declared in your Pascal source code, you can call it like any other Pascal routine.

   For example:

   ```
   BEGIN
     (* ... *)
     total := SumInCPP(subtotal1, subtotal2);
     (* ... *)
   END
   ```

   To determine how to match C types to Pascal types, follow the guidelines shown in "Matching C/C++ and Pascal Types" on page 36.

Restrictions When Calling C++ Routines

Some kinds of C++ routines can't be called from Pascal:

- Pascal can't call C++ overloaded functions. The number of parameters, their types, and the return type in the routine declared in Pascal and the routine defined in C++ through the `extern "C"` directive must match. See "Matching C/C++ and Pascal Types" on page 36.

Matching C/C++ and Pascal Types

The table below indicates what data types should be used to ensure error-free parameter passing when calling a C or C++ routine from your CodeWarrior Pascal program.

Table 3.1 C-compatible Pascal types

C/C++ type	Pascal type
`boolean (typedef)`	BOOLEAN (Note: Only function result values of zero or one will be interpreted correctly by Pascal. Some C functions may return other values.)
`enum`	enumerated (Note: If the enumerated type does not have contiguous values, "dummy" values will need to be declared in Pascal.)
`char`	–128..127 subrange (Note: If the intention is to denote the extended Macintosh character set, the result may have to be "normalized" by adding +256 to negative values. One would also do this to convert to the Pascal type CHAR.)
`short`	`CHAR`
`short`	`INTEGER`
`unsigned short`	`UNSIGNEDWORD`
`float`	`REAL`
`double`	`DOUBLE`
x`*`	`VAR` of *X'* or pointer to *X'*, where *X'* is the Pascal type corresponding to the C type *x*. Also may convert to pointer to array of *X'* when C intends an array to be passed.
`short*`	pointer to `ARRAY OF INTEGER`

`long`	`LONGINT`
`unsigned long`	`UNSIGNEDLONG`
`struct`	`RECORD`
`union`	variant `RECORD`
`array`	`ARRAY` (Note: C function results of type `ARRAY` are not allowed.)
`unsigned char`	`SET OF 0..7` (or byte size)
`unsigned short`	`UNSIGNEDWORD` `SET OF 0..15` (or word size)
`unsigned long`	`UNSIGNEDLONG` `SET OF 0..31` (or long size) For PowerPC: `PACKED ARRAY [0 .. 3] OF CHAR`

NOTE: Use of the reserved word STRING in Pascal declarations of C functions is not supported.

Examples

In C

```
long scani(char t);
```

is declared in Pascal as

```
TYPE signedByte = -128 .. 127;
FUNCTION scani(
    t : signedByte) : LONGINT; C; EXTERNAL;
```

In C

```
char expos(double d, long* i);
```

is declared in Pascal as

```
TYPE signedByte = -128 .. 127;
FUNCTION expos(
```

```
d : DOUBLE;
VAR i : LONGINT) : signedByte; C; EXTERNAL
```

4

Directives and Predefined Symbols

These topics describe how to use compiler directives, pragmas and predefined symbols to control source code translation and object code generation at compile time.

Directives and Predefined Symbols Overview

This chapter is a reference to all of the compiler directives, C-style pragmas and predefined symbols recognized by the CodeWarrior Pascal compiler.

The sections in this chapter are:

- Compiler Directives
- Pragmas
- Checking Options
- Predefined Symbols

Compiler Directives

Use compiler directives to control the source code that the compiler translates. Typically, compiler directives only have effect at compile time, not at runtime.

There are directives to control which source code the compiler translates, examine the compiler time environment, and control what kind of object code the compiler produces.

See "Using Compiler Directives" on page 40 to learn how to use compiler directives.

Refer to these topics to find out about a specific compiler directive:

$ALIGN	$D	$DEFINE
$DEFINEC	$ELIF	$ELIFC
$ELSE	$ELSEC	$ENDIF
$ENDC	$ERROR	$ERRORC
$EXPORT	$I	$IF
$IFC	$IFDEF	$IFNDEF
$IFNOTC	$INCLUDE	$J
$MAIN	$LIBEXPORT	$MC68881
$MC68881	$MC68020	$NR
$OV	$PARAM	$PARAMSTR
$POP	$PRAGMA	$PRAGMAC
$PROF	$PUSH	$R
$S	$SETC	$SC
$U	$UNDEF	$UNDEFC
$UNDEFINED	$W	$Z

Using Compiler Directives

Every compiler directive must be enclosed within comment delimiters (`{` and `}`, or `(*` and `*)`) and begins with a dollar sign (`$`). Only one directive may be within each pair of comment delimiters.

Listing 4.1 shows an example of using a compiler directive to insert the contents of a source code file into another source code file.

Listing 4.1 Example of using a compiler directive

```
program IUseInclude;

{$I GlobalDefinitions.p}
```

```
var
  x : real;
begin
  x := pi; (* Defined in GlobalDefinitions.p *)
end.
```

CodeWarrior Pascal lets you compile sections of your source text conditionally by means of the $IFC, $ELSEC, and $ENDC and associated directives. The $IFC directive is controlled by the value of a *compile-time expression*. You set such an expression using $SETC. Sections of source code controlled by these directives may be nested. The rules covering such nesting are the same as the rules for IF statements. See the *Pascal Language Reference* for more information about IF statements.

You can form compile-time expressions out of *compile-time variables* or constants of type INTEGER or BOOLEAN. The final value of the expression that controls a $IFC directive must be BOOLEAN. You can use all the Pascal operators in compile-time expressions except IN and @. If you use the operator /, the compiler will automatically change it to DIV.

$ALIGN

```
{$ALIGN POWERPC}
{$ALIGN MAC68K}
{$ALIGN RESET}
```

Determines how the compiler aligns structured data.

The $ALIGN POWERPC directive instructs CodeWarrior Pascal to align items in a data structure on their natural boundaries—the standard alignment for Mac OS PowerPC computers. It aligns 1-byte items on 1-byte boundaries, 2-byte items on 2-byte boundaries, and so on.

The $ALIGN MAC68K directive instructs the compiler to align items that are 2 bytes or larger on 2-byte boundaries. Items of 1 byte are

aligned on 1-byte boundaries. This is the standard alignment for Mac OS 68K computers.

The `$ALIGN RESET` directive instructs the compiler to restore the alignment to the previous scheme.

This directive is only available when compiling for Mac OS.

$D

```
{$D+}
{$D-}
```

Generates additional information for Apple MacsBug. The `{$D-}` directive instructs the compiler not to generate MacsBug information.

When generating PowerPC object code, `{$D+}` also generates traceback tables. `{$D-}` turns off traceback table generation.

This directive is only available when compiling for Mac OS.

$DEFINE

```
{$DEFINE id }
```

Defines an identifier that may be tested and undefined with other directives.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

See also $IFDEF, $ELSE, $ENDIF.

$DEFINEC

```
{$DEFINEC id}
{$DEFINEC id definition}
{$DEFINEC id ( parameters )definition
```

Defines an item, *id*. The `$DEFINEC` directive substitutes every occurrence of *id* in the source code with its definition, *definition*. The identifier, id, is not substituted when it appears in between comment delimiters or a string literal.

The definition part of a `$DEFINEC` directive can be any keyword, constant value, or statement that is legal in CodeWarrior Pascal. Multiple statements in the definition must be separated by semi-colons (`;`).

The item *id* may have the same name of an already-declared procedure or function if *id* and the routine don't have the same number of parameters.

If *id* is defined with *parameters*, every occurrence of a parameter is replace with the corresponding items in *id*'s definition before *id* is substituted. Parameters must be enclosed in parentheses in the `$DEFINE` directive.

Items defined in the `INTERFACE` part of a unit are accessible by the units or the program that use the unit. Items defined in the `IMPLEMENTATION` part of a unit are only accessible within the unit.

An item defined without a definition and without parameters is still defined but has an empty definition. To remove the definition of an item defined with `$DEFINEC`, use the `$UNDEFC` directive.

Listing 4.2 lists examples of using the `$DEFINEC` directive.

Listing 4.2 $DEFINEC examples

```
PROGRAM DefineExamples;

{$DEFINEC EraseMe}
{$DEFINEC two 2}
{$DEFINEC PRINT(a) WriteLn(a);}

begin
  WriteLn(two);
  EraseMe;
```

```
  PRINT('Hello, world!');
end.
```

This is a CodeWarrior Pascal-compatible directive. It may not be used in the same source code file as other Turbo-compatible directives.

See also $IFC, $IFNOTC, $UNDEFC.

$ELIF

{$ELIF *comp-expr*}

This directive is equivalent to the $ELIFC directive.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

$ELIFC

{$ELIFC *comp-expr*}

Marks the beginning of source text that is compiled only if the value of the *comp-expr* controlling its corresponding `$IFC` or `$IFNOTC` directive is `false` and its own *comp-expr* expression is true.

This is a CodeWarrior Pascal-compatible directive. It may not be used in the same source code file as other Turbo-compatible directives.

See also $IFC, $ENDC, $SETC.

$ELSE

{$ELSE}

This directive is equivalent to the $ELSEC directive.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

$ELSEC

```
{$ELSEC}
```

Marks the beginning of source text that is compiled only if the value of the *comp-expr* controlling its corresponding `$IFC` or `$IFNOTC` directive is `false`.

This is a CodeWarrior Pascal-compatible directive. It may not be used in the same source code file as other Turbo-compatible directives.

See also $IFC, $ELIFC, $ENDC, $SETC.

$ENDIF

```
{$ENDIF}
```

This directive is equivalent to the `$ENDC` directive.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

$ENDC

```
{$ENDC}
```

The `$ENDC` directive marks the end of a section of conditionally compiled source text, matching a `$IFC` or `$IFNOTC` directive.

This is a CodeWarrior Pascal-compatible directive. It may not be used in the same source code file as other Turbo-compatible directives.

See also: $IFC, $IFNOTC, $ELSEC, $SETC.

$ERROR

```
{$ERROR comment}
```

This directive is equivalent to the $ERRORC directive.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

$ERRORC

```
{$ERRORC comment}
```

Forces a compiler error. A comment, *comment*, may optionally follow this directive.

This is a CodeWarrior Pascal-compatible directive. It may not be used in the same source code file as other Turbo-compatible directives.

$EXPORT

```
{$EXPORT+}
{$EXPORT-}
{$EXPORT identifier [, . . .]}
```

Exports symbols when creating a shared library. The symbols to export, variables and routines, must be global. Before using $EXPORT directives, turn on exporting with the {$LIBEXPORT+} directive.

{$EXPORT+} tells the compiler to export all global items following this directive.

{$EXPORT-} tells the compiler not to export subsequent global items.

{$EXPORT *identifier* [, . . .]} tells the compiler to export only the specified items.

This directive is only available when compiling for Mac OS.

See also $LIBEXPORT.

$I

```
{$I filename}
```

Instructs the compiler to fetch subsequent source code input from the specified file. The compiler will read from the new file, *filename*, until the end of that file, at which point the compiler will continue from the original source file.

This is a CodeWarrior Pascal-compatible directive. It may not be used in the same source code file as other Turbo-compatible directives.

See also $U.

$IF

```
{$IF comp-expr}
{$IF OPTION(option-name) }
{$IF OPTION option-name }
```

This directive is equivalent to the `{$IFC` *comp-expr*`}` form of the `$IFC` directive.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

See also $IFDEF, $ELSE, $ELIF, $ENDIF.

$IFC

```
{$IFC comp-expr}
{$IFC DEFINED name}
{$IFC OPTION(option-name) }
{$IFC OPTION option-name }
```

Compiles subsequent source code until the next `$ELSEC` or `$ENDC` directive, only if the boolean value of *comp-expr* is `true`.

You may also write this directive in the form `{$IFC DEFINED name}`. This will act like `{$IFC false}` if *name* has not yet been declared with a `$SETC` directive; otherwise, it will act like `{$IFC true}`.

You can also write this directive `{$IFC OPTION(option-name}` to test the current setting of compiler directives. For example, `{$IFC OPTION (PROF)}` acts like `{$IFC true}` if `{$PROF+}` had been specified prior to the `$IFC`.

This is a CodeWarrior Pascal-compatible directive. It may not be used in the same source code file as other Turbo-compatible directives.

See also $IFNOTC, $ELIFC, $ELSEC, $ENDC, $SETC, $DEFINEC, "Checking Options" on page 98.

$IFDEF

`{$IFDEF` *id*`}`

This directive is equivalent to the `{$IFC DEFINED id}` form of the $IFC directive.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

See also $DEFINE, $UNDEF, $IFNDEF, $IF.

$IFOPT

`{$IFOPT` *option-name*`}`

This directive is equivalent to the

`$IF OPTION(`*option-name*`)`

form of the $IF directive.

See also $IF, $ELSE, $ELIF, $ENDIF, "Checking Options" on page 98
.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

$IFNDEF

```
{$IFDEF id }
```

This directive is equivalent to the {$IFNOTC DEFINED id} form of the $IFNOTC directive.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

See also $IFDEF.

$IFNOTC

```
{$IFNOTC DEFINED name}
```

Compiles subsequent source code until the next $ELSEC or $ENDC directive, only if the boolean value of *comp-expr* is true.

This directive will act like {$IFC true} if *name* has not yet been declared with a $SETC directive; otherwise, it will act like {$IFC false}.

This is a CodeWarrior Pascal-compatible directive. It may not be used in the same source code file as other Turbo-compatible directives.

See also: $ELSEC, $ENDC, $IFC, $SETC, $DEFINEC.

$INCLUDE

```
{$INCLUDE file}
```

This directive is equivalent to the $I directive.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

$J

```
{$J+}
{$J-}
```

Tells to the compiler that one or more variables *declared* in your this source file are not *defined* in this source file, but somewhere else, such as a linked C routine.

Variable declarations you want the compiler to treat this way should be preceded by `{J+}` and followed by `{J-}`, or preceded by `{$PUSH}{$J+}` and followed by a `{$POP}`.

This directive also controls the case conversion of global identifiers.

Normally, the global identifiers are converted to all uppercase to conform to the 68K MPW calling conventions. With `{$J+}`, no case conversion is done, preserving the original form of global identifiers in the object code. Use the $J directive to preserve identifier case when calling C global variables from Pascal, and calling Pascal global variables from C.

The default setting of this directive is `{$J-}.`

This directive is only available when compiling for Mac OS.

$LIBEXPORT

```
{$LIBEXPORT+}
{$LIBEXPORT-}
```

To begin exporting symbols when creating a shared library, use `{$LIBEXPORT+}` followed by `{$EXPORT . . .}` directives to indicate which symbols to export.

To stop exporting symbols in a shared library, use `{$LIBEXPORT-}`.

This directive is only available when compiling for Mac OS.

$MAIN

```
{$MAIN}
```

For 68000-family object code generation only. The `$MAIN` directive instructs CodeWarrior Pascal to make the following routine the program entry point. This directive is used to create code resources.

Code resources can only be built from units. Place the `$MAIN` directive before the definition of the entry point routine, in the unit's interface part, the main entry point.

This directive is only available when compiling for Mac OS.

$MACRO

```
{$MACRO id}
{$MACRO id definition}
{$MACRO id ( parameters )definition
```

This directive is equivalent to the $DEFINEC directive.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

$MC68020

```
{$MC68020+}
{$MC68020–}
```

For code generation for the Motorola 68000 family of microprocessors only. The `$MC68020+` directive causes the generation of object code for the 68020 object code. The `$MC68020–` directive causes the generation of object code for the 68000.

This directive is equivalent to the 68020 codegen option in the 68K Processor preferences panel in the Preferences dialog box. This directive has no effect for PowerPC code generation.

NOTE: Using the `$MC68020+` directive excludes your object code from executing on the 68000 processor.

This directive is only available when compiling for Mac OS.

$MC68881

```
{$MC68881+}
{$MC68881-}
```

For code generation for the Motorola 68000 family of microprocessors only. The `$MC68881+` directive causes floating point operations to make direct calls to the 68881 and 68882 floating point coprocessors and floating point unit built-in to the 68040. The `$MC68881-` directive causes floating point operations to be computed in software.

This directive is equivalent to the 68881 codegen option in the 68K Processor preferences panel in the Preferences dialog box. This directive has no effect for PowerPC code generation.

NOTE: Using the `$MC68881+` directive excludes your object code from running on 68K Macintosh computers without hardware floating point capabilities.

This directive is only available when compiling for Mac OS.

$NR

```
{$NR+}
{$NR-}
```

Turns subsequent function calls into procedure calls. The `$NR-` directive returns the code to normal behavior.

Listing 4.3 Using the $NR Directive

```
function foo: integer; begin end;
...
begin
```

```
{$NR+}
foo;
{NR-}
```

$OV

```
{$OV+}
{$OV-}
```

The default condition `$OV-` prevents the compiler from generating code to detect arithmetic overflow during assignments and expression evaluation. The directive `$OV+` causes it to produce such code.

This directive is equivalent to the **Activate Overflow Checking** option in the **Pascal Language Settings** panel.

$PARAMSTR

```
{$paramstr+}
{$paramstr-}
```

Checks for possible memory mangling in string parameters when the actual and formal types are not the same size. Using `{$paramstr+}` allows parameter string size comparison, and using `{$paramstr-}` disables this comparison.

This directive is equivalent to the **Check String Parameter Sizes** option in the **Pascal Warnings** settings panel.

$PARAM

Removes parameter names when compiling units. Using this directive significantly reduces the size of the precompiled unit.

Place this directive in the `INTERFACE` part of a unit, before any routines or object methods are defined. Make sure the routines in the `IMPLEMENTATION` part of the unit re-declare the parameters.

$POP

{$POP}

Restores compiler directive settings. Use the $PUSH and `$POP` directives to save and restore directive settings temporarily when they are modified for a special case in source code.

The directive settings that `$POP` restores are

- `$R`
- `$OV`
- `$SC`
- `$I`
- `$Z`
- `$PROF`

$PRAGMA

`{$PRAGMA` *pragma state*`}`

This directive is equivalent to the `$PRAGMAC` directive.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

$PRAGMAC

`{$PRAGMA` *pragma state*`}`

Sets the state of a pragma. See "Pragmas" on page 60 for more information on using pragmas to control how source code is translated into object code.

This is a CodeWarrior Pascal-compatible directive. It may not be used in the same source code file as other Turbo-compatible directives.

$PROF

```
{$PROF+}
{$PROF-}
```

Generates object code to use with the CodeWarrior Profiler. With `{$PROF+}` placed at the beginning of your source code, the profiler can measure the time spent in routines and how often routines are called. The `{$PROF-}` statement turns profiler code generation off.

Consult the *CodeWarrior Profiler Manual* for information on which libraries to add to your project and how to use the profiler viewer.

This directive is only available when compiling for Mac OS.

$PUSH

```
{$PUSH}
```

Saves compiler directive settings. Use the `$PUSH` and $POP directives to save and restore directive settings temporarily when they are modified for a special case in source code.

The directive settings that `$PUSH` saves are

- $R
- $OV
- $SC
- $I
- $Z
- $PROF

These directives may be saved up to ten levels deep.

$R

```
{$R+}
{$R-}
```

Generates code to perform range checking of string, set, and array bounds.

This directive is equivalent to the **Activate Range Checking** option in the **Pascal Language Settings** panel.

$S

```
{$S [segname]}
```

Specifies code segment organization.

By default, the compiler will instruct the linker to place all routines within a single segment (with the case-sensitive identifier `Main`). Use the `$S` directive to specify that subsequent routines be directed to the specified segment. The `$S` command can only appear between global routines. If *segname* is omitted, the segment name `Main` is assumed.

A segment is a part of code that can be loaded separately into memory. Your program can be written without explicit segmentation or it can contain a number of different segments.

Each '`CODE`' resource in the application's resource fork corresponds to a segment containing one or more routines. (The '`CODE`' resource with ID 0 contains the jump table to other code resources; other '`CODE`' resources contain executable routines). At run time, a segment is automatically loaded by the Segment Loader when you call one of the routines contained in the segment. The segment is not unloaded until the application explicitly unloads it by calling `UnloadSeg`. See *Inside Macintosh* for more information about the Segment Loader.

You can specify which routines are placed in which segments in two ways:

- use the `$S` directive in your Pascal source code to specify segmentation
- use the project window to arrange a program's segmentation

Segmentation helps you reduce your program's run-time memory requirements. A typical segmentation scheme divides a program into an initialization segment and a main processing segment. You can also put routines that are seldom executed—printing routines, for instance—in a separate segment that is not loaded when the pro-

gram begins executing. This allows the program to be loaded faster since printing routines are not loaded until they are needed. If you don't specify segmentation, the compiler puts the entire program into a segment called `Main`.

The `$S` directive also lets you specify several segments within a single source file. To assign source code to a segment, precede the code with a compiler directive of the form

```
{$S segname}
```

The code following this directive is placed in the named segment until the compiler reads another `$S` or the end of the source file.

NOTE: In an `$S` directive, segment names are case sensitive. Leading spaces are not significant, and all characters are included, up to the end or the comment character.

Code for a given segment does not have to be contiguous within the source file. The program may take the following form:

```
{$S SegA}
function
{$S SegB}
function
{$S SegA}
```

and so forth.

The compiler marks each routine with the name of its segment. Then the linker collects all of the functions and procedures for a segment and places them into one code segment in the linked executable file.

The *CodeWarrior IDE User Guide* explains how to use the project window to arrange a program's segmentation.

This directive only applies to CodeWarrior Pascal programs targeted for Mac OS 68K. CodeWarrior ignores this directive when targeting all other platforms.

$SETC

```
{$SETC id := comp-expr}
```

Declares a compile-time variable named *id* and assigns it the value *comp-expr*.

This directive may not be used with Turbo Pascal-compatible directives.

See also: $IFC, $ELSEC, $ENDC.

$SC

```
{$SC+}
{$SC-}
```

Evaluates `AND` and `OR` as short-circuit operators. In this case, the evaluation process starts with the left operand and ends when the evaluation of the expression is forced. The default directive `$SC-` causes the compiler to evaluate all operands of `AND` and `OR`.

For example, with `{$SC+}`, the expression

```
(false OR false) AND (true OR true)
```

stops before the second operand of the middle `AND` operator, `(true OR true)`, is reached because the value of the expression is already known when the first operand, `(false OR false)` is evaluated. In other words, for AND to give a `true` value, both of its operands must be `true`. Because the first operand, `(false OR false)` is `false`, the second operand, `(true OR true)` does not have to be evaluated—the program already knows that the AND operator will give a `false` result.

$U

```
{$U unitname}
```

Overrides the default name for a unit in a USES statement. In a `USES` statement, the compiler normally expects a unit name to be in a file *unitname*.p. This mechanism can be overridden by using the `$U`

command to specify a filename for the following unit. In either case, the compiler will use its access rules to open the file. Each `$U` directive is valid only once for the next unit name specified in the source text.

$UNDEF

```
{$UNDEF id}
```

This directive is equivalent to the `$UNDEFC` directive.

This is a Turbo Pascal-compatible directive. It may not be used in the same source code file as other CodeWarrior-compatible directives.

See also $DEFINE.

$UNDEFC

```
{$UNDEFC id}
```

Undefines an identifier that was defined with the `$DEFINEC` directive. Undefining a previously defined item tells the compiler not to substitute subsequent occurrences of *id* with its definition.

Use this directive with the `$IFC` and `$IFNOTC` directives to control the compilation of source code.

This is a CodeWarrior Pascal-compatible directive. It may not be used in the same source code file as other Turbo-compatible directives.

See also $DEFINEC, $IFC, $IFNOTC.

$UNDEFINED

```
{$UNDEFINED proc, proc}
```

Disables warning messages for the listed routines.

$UNUSED

```
{$UNUSED identifier [, ...]}
```

Tells the compiler not to issue an Unused Arguments warning or Unused Variables warning for the specified parameters and variables if those parameters and variables are unused in the current scope.

The `$UNUSED` directive must be within a routine's scope.

$W

```
{$W+}
{$W-}
```

Controls peephole optimization. To turn off peephole optimization, use the `{$W-}`. This directive applies for all platforms.

$Z

```
{$Z+}
{$Z-}
{$Z*}
```

Makes identifiers accessible outside a unit as if they were declared in the interface part. Other units can use identifiers declared following `$Z+` by declaring them as `EXTERNAL`.

Identifiers following the `$Z-` directive are not accessible outside the unit unless they are declared in the interface part.

The `$Z*` directive, a subset of `$Z+`, forces the compiler to identify subsequent routines at the top nesting level (and not variables) as external.

Pragmas

CodeWarrior Pascal allows your source code to change how the compiler compiles it with pragmas. Many of the pragmas correspond to options in the **Pascal Language Settings** panel.

These topics show you how to use pragmas:

- Using Pragmas
- Pragma Syntax

These topics describe the pragmas:

a6frames	align	casesensitive
ANS_conformance	chkstrparam	code68881
code68020	codelisting	codesize
codeviewinfo	copyparam	d0_pointers
deadcode	deadstore	debuglisting
disable_register	dont_reuse_sets	dont_reuse_strings
disable_register	expandtables	export
external	externalvar	fardata
far_code	farmtables	farstrings
force_active	fourbyteints	fp_contract
global_optimizer	IEEEdoubles	import
inlinedispatching	internal	intrinsic
IOmodel	lifetime	logopt
lib_export	macsbug	makemap
mark	mpwc	near_code
noreturn	objcheck	objectPascal
oldstyle_symbols	optimizeclasses	optimize_for_size
optimize_for_space	optimize_for_speed	overflow
pack	pcrelstrings	peephole
poolsets	poolstrings	pop
precompile_target	profile	propagation
ptrbasedobject	push	rangecheck

readonlystrings	regcoloring	relaxedptrcheck
scheduling	shortcircuit	sideeffects
smart_code	strengthreduction	strsize
subexpression	sym	syminfo
toc_data	traceback	trapnil
trapnil	usespropagation	warn_foridx
warn_gotocase	warn_gotoif	warn_gotofor
warn_gotowith	warn_illpragma	warn_noreturn
warn_undefroutine	warn_unusedarg	warn_unusedvar

Using Pragmas

NOTE: Some pragmas may not apply to your target. To know which ones do apply, see the chapter for your platform in this guide.

You'll usually use the **Pascal Language Settings** panel to set the options for most of your code and use pragmas to change the options for special cases. For example, with the Project Settings dialog, you can turn off a time-consuming optimization and, with a pragma, turn it on only for the code it helps most.

This section assumes that you are generally familiar with the use of pragmas and Pascal. As a reminder, a line of the form

```
$pragma option-name [switches]
```

causes the CodeWarrior Pascal compiler to change its behavior when the option name is one recognized by the compiler. Unrecognized pragmas are ignored.

See also $PRAGMA.

Pragma Syntax

Pascal pragmas use the following syntax:

```
$pragma option-name on | off | reset
```

Generally, use on or `off` to change the option's setting, and then use `reset` to restore the option's original setting, as shown in the code example below:

```
$pragma rangecheck off { If the option Range Check is on, turn it
                         off for these functions.}
Const
  Ceiling = 10;

(* ... *)

for Index:= 1 to Ceiling do
  Writeln(Index:1);
Writeln('Loop has terminated.');

$pragma rangecheck reset  { If the option Range Check was on,
                            turn it back on. Otherwise, the
                            option remains off.}
```

Suppose that you use `$pragma rangecheck on` instead of `$pragma rangecheck reset`. If the setting coming into this code was originally off, `$pragma rangecheck on` turns it on for all subsequent code. Using `reset` ensures that you don't inadvertently change the setting.

See also $PRAGMA.

a6frames

```
$pragma a6frames on | off | reset
```

Generates A6 stack frames. Many debuggers, including the CodeWarrior debugger, Jasik's The Debugger, and SourceBug™, require A6 stack frames, which let debuggers trace through the call

stack and find each routine. If you are not planning to use a debugger, turn this pragma off to generate faster code.

lists the assembly language code that the compiler generates for each function, if this pragma is turned on.

Listing 4.4 A6 stack frame code generation

```
LINK #nn, A6
; executable object code for the routine goes here
UNLK A6
```

This pragma is equivalent to the **Generate A6 Stack Frames** option In the **68K Linker** settings panel.

This pragma is available for 68K programming only.

align

```
$pragma align MAC68K | MAC68K4BYTE | POWERPC |
reset
```

Organizes records and objects.

If you choose...	The compiler aligns every field on...
MAC68K	2-byte boundaries, unless a field is only 1-byte long. This is the standard alignment for 68K Macintosh computers.

If you choose...	The compiler aligns every field on...
MAC68K4BYTE	4-byte boundaries.
POWERPC	This is the standard alignment for Power Macintosh computers. For example, it aligns a character on a 1-byte boundary and a 16-bit integer on a 2-byte boundary. The compiler applies this alignment recursively to structured data and arrays containing structured data. So, for example, it aligns an array of structured types containing an 8-byte floating point member on an 8-byte boundary.

This pragma corresponds to the **Struct Alignment option** in the **68K Processor** and **PPC Processor** preferences panels. The default setting of this pragma is 68K.

This pragma is available for 68K and PowerPC programming only.

ANS_conformance

```
$pragma ANS_conformance on | off | reset
```

Generates an error when it encounters certain ANS features that conflict with the ANS Pascal specification (see *The American Pascal Standard, With Annotations* for additional information on ANS Pascal). Use this option only if you must make sure that your code strictly follows the ANS specification.

This pragma corresponds to the **ANS Conformance** option in the **Pascal Language Settings** panel.

casesensitive

```
$pragma casesensitive on | off | reset
```

Uses case sensitive comparison for identifiers. This pragma corresponds to the **Case Sensitive Identifiers** option in the **Pascal Language Settings** panel.

chkstrparam

```
$pragma chkstrparam on | off | reset
```

Checks declared string sizes and generates a warning for cases where memory mangling is likely. This warning is useful because string type compatibility doesn't take length into account. This can produce hard-to-find bugs when you pass by reference a shorter string to a longer one.

Turning this pragma on is equivalent to checking the **Check String Parameter Sizes** option in the Pascal Warnings settings panel.

code68020

```
$pragma code68020 on | off | reset
```

Uses the extensions available in the MC68020 instruction set, including integer multiplication, integer division, and bit-field operations. These extensions are available in the MC68030 and MC68040 as well.

WARNING! Before your program runs code optimized for the MC68020, use the gestalt() function to make sure it is available. For more information on gestalt(), see Chapter "Gestalt Manager" in Inside Macintosh: Operating System Utilities.

This pragma is equivalent to the **68020 Codegen** option in the **68K Processor** settings panel.

This pragma is available for 68K programming only.

code68881

```
$pragma code68881 on | off | reset
```

Generates MC68881 instructions to perform floating point operations. The MC68881 is built into most versions of the MC68040 processor, and it is included separately in many Macintosh computers that contain the MC68030 or MC68020 processor.

CodeWarrior C/C++ and Pascal give you two levels of support for the MC68881, as described below:

- No matter what you do, the Macintosh Toolbox uses the MC68881 for many floating-point functions.
- If you also turn this pragma on, the compiler generates code optimized for the MC68881 and stores variables declared long double or extended in 96 bits. It uses MC68881 instructions for basic arithmetic operations, such as addition, subtraction, multiplication, division, and comparisons. The compiled code is faster and computes the same results as code compiled with the option off.

Think carefully before you use this pragma. Your code will not run on a PowerPC or any 68K Mac OS computer that does not have an MC68881. And remember, even if you do not use this pragma, the Mac OS toolbox will use the floating-point unit to compute many floating-point functions.

WARNING! Before your Mac OS program runs code optimized for the MC68881, use the `gestalt()` function to make sure the FPU is available. For more information on `gestalt()`, see Chapter "Gestalt Manager" in Inside Macintosh: Operating System Utilities.

This pragma corresponds to the **68881 Codegen** option in the **68K Processor** preferences panel. Its default setting is off.

This pragma is available for 68K programming only.

codelisting

```
$pragma codelisting on | off | reset
```

Creates a disassembly of the object code generated by the compiler.

This option is equivalent to the **Machine Code Listing** option in the **x86 Codegen** settings panel.

This pragma is available for x86 programming only.

codesize

```
$pragma codesize far | near | smart | reset
```

Specifies what kind of addressing to use. This pragma is equivalent to the **Code Model** option in the **68K Processor** preferences panel, and indicates to the compiler how it should deal with addressing:

- Smart is best for most cases. When one function calls another function and the linker is sure they're in the same segment, this model uses relative addressing (a 16-bit address). Otherwise, it uses absolute addressing (a 32-bit address).
- Near is best for small code resources and applications (under 32K), or for code resources and applications in which all segments are under 32K.This model uses relative addressing (a 16-bit address) for all function calls known to be within 32K.
- Far is useful when you have a source file that generates more than 32K of code or when CodeWarrior generates an out-of-range link error. This model uses absolute addressing (a 32-bit address) for all function calls.

The default setting for this pragma is smart.

This pragma is available for 68K programming only.

codeviewinfo

```
$pragma codeviewinfo on | off | reset
```

Turning this pragma on instructs the compiler to generate symbolic information in CodeView format.

NOTE: You must turn on both CodeView format and Generate CodeView Info on the x86 Linker panel to create CodeView symbolics for debugging.

This pragma is equivalent to the **CodeView Format** option in the **x86 CodeGen** preferences panel. Its default setting is off.

copyparam

```
$pragma copyparam on | off | reset
```

Optimizes parameter passing by deleting unnecessary copies of structured type value parameters.

This pragma corresponds to the **Copy Value Parameter** option in the **Pascal Language Settings** panel.

d0_pointers

```
$pragma d0_pointers on | off | reset
```

Returns function pointer results in the D0 register; normally they are returned in A0.

The default setting of this pragma is off.

This pragma is available for 68K programming only.

deadcode

```
$pragma deadcode on | off | reset
```

Looks for and remove sections of code that can never be reached in the course of program execution. This reduces code size, but does not affect execution speed.

This pragma is equivalent to the **Dead Code Elimination** option in the **Global Optimizations** settings panel.

deadstore

```
$pragma deadstore on | off | reset
```

Remove assignments to variables whose values are not used again before being reassigned, saving both execution time and code size.

This pragma is equivalent to the **Dead Store Elimination** option in the **Global Optimizations** settings panel.

debuglisting

```
$pragma debuglisting on | off | reset
```

Generates a text listing of global optimizations that were applied during compilation.

This pragma is available for PowerPC programming only.

disable_register

```
$pragma disable_register on | off | reset
```

Turning this pragma on causes the compiler to disable register allocation/optimization in any routine that makes non-local gotos.

The default setting for this pragma is off.

dont_reuse_sets

```
$pragma dont_reuse_sets on | off | reset
```

Stores each set literal separately, allowing multiple copies of set constructors per object file. If this pragma is off, the compiler stores only one copy of identical set literals. This pragma helps you save memory if your program contains many identical set literals which you do not modify.

This pragma corresponds to the **Don't Reuse Sets** option in the **Pascal Language Settings** panel.

dont_reuse_strings

```
$pragma dont_reuse_strings on | off | reset
```

Stores each string literal separately. When this pragma is set to `off`, the compiler stores only one copy of identical string literals. This option helps you save memory if your program contains lots of identical string literals which you do not modify.

This pragma is equivalent to the **Don't Reuse Strings** option in the **Pascal Language Settings** panel.

expandtables

```
$pragma expandtables on | off | reset
```

Generates full method tables. Turning this option on increases the speed of your application and data size by improving calling performance, but it causes your application to use more memory. This pragma must be set for the entire class hierarchy for it to work properly.

This pragma corresponds to the **Expand Method Tables** option in the Pascal Language preferences panel.

This pragma is available for 68K and PowerPC programming only.

export

```
$pragma export on | off | reset | list [export-
list]
```

Makes normally internal public symbols external to your module. There are two ways to use this pragma:

- By preceding one or more function or procedure declarations in your code with `$pragma export on`, and following them with `$pragma export off`. All symbols defined or declared while the export pragma is in the `on` state are exported.
- By simply using `$pragma export` *export-list*, where `export-list` is the list of the symbols you want to export, separated by spaces.

The export pragma should always be used together with a setting of `Use $pragma` for `Export Symbols`, in the CodeWarrior IDE's Target Settings preferences panels. To display the Export Symbols setting:

- Select PPC PEF under Linker when targeting PPC

external

```
$pragma external on | off | reset
```

Declares items as global. When this pragma is on, subsequent variables and routines are have global scope. When this pragma is off, subsequent variables and routines have normal scope.

externalvar

```
$pragma externalvar on | off | reset
```

Declares items as global with C linkage conventions. When this pragma is on, subsequent variables and routines are have global scope and use C linkage conventions. When this pragma is off, subsequent variables and routines have normal scope and use Pascal linkage conventions.

far_code

```
$pragma far_code
```

This is simply a shortcut to setting the codesize pragma to far, like this:

```
$pragma codesize far
```

This pragma is available for 68K programming only.

See "codesize" on page 68 for more information.

fardata

```
$pragma fardata on | off | reset
```

Uses addressing to access a (practically) unlimited amount of global memory by storing it in far data space. However, the compiled code will be slightly larger and slower.

This pragma corresponds to the **Far Data** option in the **68K Processor** preferences panel. The default setting for this pragma is off.

This pragma is available for 68K programming only.

farmtables

```
$pragma farmtables on | off | reset
```

Generates method tables that are referred to using "far" addressing.

If you use Object Pascal language elements in a project, the linker needs to store method dispatch tables with the global data. When your application is running and calls a method, it uses these tables to find the address of the appropriate implementation of that method.

If your project is large or uses an exceptional number of classes, the method dispatch tables could overflow the 64K usually reserved for global data. Turn this pragma on to store the these tables in far data, separate from the global data.

This pragma corresponds to the **Far Method Tables** option in the **68K Processor** preferences panel. The default setting for this pragma is off.

This pragma is available for 68K programming only.

farstrings

```
$pragma farstrings on | off | reset
```

Stores all string constants in near data space. If you turn this pragma on, the linker stores all string constants with the rest of the global data in far data space. Turning this pragma on therefore gives you more room for strings in your project, but at the expense of a reduction in speed and an increase in size.

This pragma corresponds to the **Far String Constants** option in the **68K Processor** preferences panel. The default setting for this pragma is off.

This pragma is available for 68K programming only.

force_active

```
$pragma force_active on | off | reset
```

Removes the machine code for unused variables and routines from the compiled-but-unlinked version of your project.

This practice by the linker is called dead-stripping, and this pragma corresponds to the **Dead-Strip Static Initialization Code** option in the **68K Linker** and **PPC Linker** preferences panels. However, it is an inverse correspondence. The on state of force_active corresponds to the Dead-Strip Static Initialization Code option not being selected, and vice-versa.

This pragma is available for 68K programming only.

force_public_data

```
$pragma force_public_data on | off | reset
```

Makes the default visibility of data members to public. When this option is on, the default visibility of data members in classes is public. When this option is off, the default is protected.

By default this pragma is on.

fourbyteints

```
$pragma fourbyteints on | off | reset
```

Gives the `integer` data type 4 bytes. When this option is on, items of type `integer` are 4 bytes. When this option is off, integers are 2 bytes.

This pragma is equivalent to the **4 Byte Ints** option in the **68K Processor** settings panel.

This pragma is available for 68K programming only.

fp_contract

```
$pragma fp_contract on | off | reset
```

Uses FMADD, FMSUB, and FNMAD instructions to improve floating point computation performance.

This pragma is equivalent to the **Use FMADD & FMSUB** option in the **PPC Processor** preferences panel. When this pragma is turned on, the compiler uses such PowerPC instructions as FMADD, FMSUB, and FNMAD to speed up floating-point computations. However, certain computations give unexpected results when this pragma is on. For example:

```
var A, B, C, D, Y, Z, T1, T2: double;
A = C = 2.0e23;
B = D = 3.0e23;
Y = (A * B) - (C * D);
writeln("Y = ",Y);
(* prints
 212677005875609618756336929996 8.000000 *)
T1 = (A * B);
T2 = (C * D);
Z = T1 - T2;
writeln("Z = ",Z);
(* prints 0.000000 *)
```

When this option is off, X and Y have the same value.

The default setting for this pragma is on.

This pragma is available for PowerPC programming only.

global_optimizer

```
$pragma global_optimizer on | off | reset
```

Turns Global Optimizations of code on and off. If turned on, there are four levels of global optimizations, and the level is selected using the optimization_level pragma. See "optimization_level" on page 82.

IEEEdoubles

```
$pragma IEEEdoubles on | off | reset
```

Sets the size of floating-point numbers for 68K projects; it affects variables declared longreal. In general, keep this pragma turned off since 8-byte (or 64-bit) floating-point numbers are less efficient than others. However, if you are porting code that relies on 8-byte floating-point numbers, you should turn this pragma on.

This pragma corresponds to the **8-Byte Doubles** option in the **68K Processor** preferences panel. The default setting for this pragma is off.

This pragma is available for 68K programming only.

import

```
$pragma import on | off | reset |
   list import-list
```

Makes available items that were exported by other files. Turning this pragma is similar to turning on the export pragma (see "export" on page 71), except that a list of symbols is imported for availability during compiling, rather than exported for importing by other projects.

The default setting of this pragma is off.

inlinedispatching

```
$pragma inlinedispatching on | off | reset
```

Generates inline method dispatches. When this pragma is on, the compiler generates object code that calls object methods directly rather than going through a method table. Although object code is faster, it may be larger.

When this pragma is off, object methods are called by referring to method tables.

This pragma is available for 68K and PowerPC programming only.

internal

```
$pragma internal on | off | reset |
   list internal-list
```

Declares items as local to a file. When this pragma is on, the scope of subsequent variables and routines is limited to the source code file. When this pragma is off, subsequent variables and routines have normal scope.

intrinsic

```
$pragma intrinsic on | off | reset
```

Generates inline object code for intrinsic functions. This option is equivalent to the **Inline Intrinsic Functions** option in **x86 Codegen** settings panel.

This pragma is available for 68K programming only.

IOmodel

```
$pragma iomodel ansIO | mpwIO | turboIO
```

Selects the input/output model your Pascal source code follows:

- If your code follows ANS Pascal standards for input/output, use `$pragma iomodel ansIO.`
- If your code follows MPW Pascal standards for input and output, use `$pragma iomodel mpwIO.`
- If your code follows the Turbo Pascal Input/Output model, use `$pragma iomodel turboIO.`

The three options of this pragma are equivalent to selecting **ANS Pascal, MPW Pascal** and **Turbo Pascal**, respectively, for **IO Model** in the **Pascal Language** preference panel.

lib_export

```
$pragma lib_export on | off | reset
```

Exports all global symbols in your module. Note that if you turn on this pragma, there is no point in using `$pragma export on` and `$pragma export off` to export a selected group of global symbols. See "export" on page 71.

Turning this pragma on is equivalent to a setting of **All globals** for **Export Symbols**, in the **Target preferences panel**s. To display the Export Symbols setting:

- Select **PPC PEF** under **Linker** when targeting PPC

lifetime

```
$pragma lifetime on | off | reset
```

Examines the use of local variables and temporaries in order to assign registers more effectively. For instance, using one register to operate on two variables whose lifetimes do not overlap.

This pragma is equivalent to the **Loop Invariants** option in the **Global Optimizations** settings panel.

logopt

```
$pragma logopt on | off | reset
```

Generates a file listing the optimizations used while compiling.

This pragma is available for 68K and x86 programming only.

loopinvariant

```
$pragma loopinvariant on | off | reset
```

Finds expressions calculated within a loop whose values don't change from one iteration to the next, and move them out of the loop so that they are evaluated only once. This reduces execution time, but does carry a small penalty in code size.

This pragma is equivalent to the **Lifetime Analysis** option in the **Global Optimizations** settings panel.

This pragma is available for 68K and x86 programming only.

macsbug

```
$pragma macsbug on | off | reset
```

Generates debugging symbols needed for the Apple MacsBug debugger in your project. If this pragma is turned on, MacsBug symbols will be included. If this pragma is turned off, MacsBug symbols will not be included. Note that this pragma only takes effect the next time CodeWarrior makes your project.

This pragma partly corresponds to the **MacsBug Symbols** option in the **68K Linker** preferences panel; it is evaluated in tandem with the `oldstyle_symbols` pragma (see "oldstyle_symbols" on page 81). The default setting of this pragma is off, which corresponds to a setting of None for **MacsBug Symbols**.

NOTE: If this pragma is turned off, the value of oldstyle_symbols is irrelevant because no MacsBug debugging symbols are included in your project.

This pragma is available for 68K programming only.

makemap

```
$pragma makemap on | off | reset
```

Generates a map file. When this pragma is on, the compiler generates a map file listing the identifiers that the compiler processed.

mark

```
$pragma mark markname
```

Inserts an item the CodeWarrior IDE function popup menu. For example, adding the line

```
$pragma mark Floogle
```

to your source code will add a new jump-point named "Floogle" to appear in the IDE function popup menu after the file is compiled. Choosing Floogle in the menu will jump to the `$pragma mark Floogle` directive.

mpwc

```
$pragma mpwc on | off | reset
```

Uses Apple's MPW C calling conventions. See the *C Compilers Reference* for information on MPW C calling conventions.

This pragma is available for 68K programming only.

near_code

```
$pragma near_code
```

This is simply a shortcut to setting the codesize pragma to far, like this:

```
$pragma codesize near
```

See "codesize" on page 68 for more information.

This pragma is available for 68K programming only.

noreturn

```
$pragma noreturn on | off | reset
```

Allows function calls to be used as routine calls. If this pragma is on, subsequent function definitions don't need to return a value.

objcheck

```
$pragma objcheck on | off | reset
```

Checks for the unsafe use of handle-based objects.

objectPascal

```
$pragma objectPascal on | off | reset
```

Makes Object Pascal language extensions available.

Setting this pragma to on is equivalent to checking the **Activate Object Pascal** check box in the **Pascal Language** preferences panel. Set it to `on`, or check the **Activate Object Pascal** check box, when your source code uses Object Pascal extensions.

The default setting for this pragma is on.

oldstyle_symbols

```
$pragma oldstyle_symbols on | off | reset
```

Generates MacsBug debugging symbol information using a format recognized by older versions of MacsBug.

The setting of this pragma is only relevant if the macsbug pragma (see "macsbug" on page 79) is set on, which causes MacsBug debugging symbols to be included in your project the next time CodeWarrior makes it. If MacsBug symbols will be included in the next make of your project, this pragma tells the compiler whether those symbols should be old-style or new-style.

This pragma is the one which, together with the macsbug pragma, corresponds to the **MacsBug Symbols** setting in the **68K Linker** preferences panel.

- A setting of off for this pragma, which is its default setting, corresponds to a setting of **New Style** for **MacsBug Symbols**.
- A setting of on for this pragma, which is its default setting, corresponds to a setting of **Old Style** for **MacsBug Symbols**.

This pragma is available for 68K programming only.

optimization_level

```
$pragma optimization_level 1 | 2 | 3 | 4
```

Determines the level of global optimizations that the compiler will perform as it compiles your project.

- **Level 1** is the simplest and includes one optimization, register coloring.
- **Level 2** is best for most code. Its optimizations include all those in Level 1 plus these:
  - Global common subexpression elimination (also called CSE)
  - Copy propagation
- **Level 3** is best for code with many loops. Its optimizations are all those in Level 2 plus these:
  - Moving invariant expressions out of loops (also called Code motion)
  - Strength reduction of induction variables
  - Using the CTR register for loops that execute a known number of times.
  - Loop unrolling.
- **Level 4** optimizes loops even more, but takes more time. Its options include all those in Level 3, plus it performs CSE and Code motion a second time, in case the loop optimizations create new opportunities.

This pragma is equivalent to the **Global Optimizations Level** option in the PPC Processor preferences panel when the **Global Optimizations** option is selected (turned on).

This pragma is available for PowerPC programming only.

optimizeclasses

```
$pragma optimizeclasses on | off | reset
```

Enables unused class stripping. This pragma must be set for the entire class hierarchy.

This pragma is equivalent to the **Optimize Class Hierarchy** option in the **Pascal Language Settings** panel. Its default setting is on if the `objectPascal` pragma is turned on. If `objectPascal` is not turned on, the setting of this pragma has no effect.

optimize_for_size

```
$pragma optimize_for_size on | off | reset
```

Prefers small code over fast code. The default setting is `off`.

This pragma is equivalent to the **Optimize Space** option in the **Global Optimizations** settings panel.

optimize_for_space

```
$pragma optimize_for_space on | off | reset
```

Prefers small code over fast code.

Turning both the `optimize_for_space` and `optimize_for_speed` settings off disables all global optimizations. This can be useful during the development cycle, when you're more interested in quick compilation than in the efficiency of the resulting code. When you do your final build, you'll probably want to turn on at least some form of code optimization. Turning both settings on performs all optimizations that can improve either space or speed without incurring a large penalty for the other. This is normally the best setting to use.

Turning one setting on and the other off optimizes for the selected factor (space or speed), even if it may adversely affect the other. You should do this only when one of the two factors is of overriding importance.

This pragma is equivalent to the **Optimize Space** option in the **Global Optimizations** settings panel.

optimize_for_speed

```
$pragma optimize_for_speed on | off | reset
```

Prefers fast code over small code; setting it off prefers small code over fast code.

Turning both the `optimize_for_space` and `optimize_for_speed` settings off disables all global optimizations. This can be useful during the development cycle, when you're more interested in quick compilation than in the efficiency of the resulting code. When you do your final build, you'll probably want to turn on at least some form of code optimization.

This pragma is equivalent to the **Optimize Speed** option in the **Global Optimizations** settings panel.

This pragma is available for 68K and x86 programming only.

overflow

```
$pragma overflow on | off | reset
```

Generates code to check for run-time overflows. Turning on this pragma increases code size, and reduces execution speed.

This option is equivalent to the **Activate Overflow Checking** option in the **Pascal Language Settings** panel.

pack

```
$pragma pack( n | push, n | pop)
```

Sets the organization of record and object structures. This pragma is equivalent to the **Byte Alignment** option in the **x86 Codegen** settings panel. The value of *n* may be 1, 2, 4, 8, or 16.

Use

```
{$pragma pack n}
```

to specify that items in records and objects should be aligned on *n*-byte boundaries.

Use

```
{$pragma pack push, n}
```

to save the current alignment value and specify that items in records and objects should be aligned on *n*-byte boundaries.

Use

```
{$pragma pack pop}
```

to restore the alignment value used in the last `{$pragma pack push, n}` directive.

Use

```
{$pragma pack}
```

to specify that items in records and objects should be aligned to the value specified in the **Byte Alignment** option in the **x86 Codegen** settings panel.

This pragma is available for x86 programming only.

pcrelstrings

```
$pragma pcrelstrings on | off | reset
```

Stores the string constants used in a local scope in the code segment and addresses these strings with instructions that use addressing relative to the program counter.

If this option is off, the linker stores all string constants in the global data segment. Regardless of how this option is set, the compiler stores string constants used in the *global* scope in the *global* data segment.

This pragma corresponds to the **PC-Relative Strings** option in the **68K Processor** preferences panel. The default setting for this pragma is off.

This pragma is available for 68K programming only.

peephole

```
$pragma peephole on | off | reset
```

Use small local optimizations that eliminate some compare instructions and improve branch sequences.

- For 68K code, this pragma to on is equivalent to the **Peephole Optimizer** option in the **68K Processor** settings panel.
- For PowerPC code, this pragma to on is equivalent to the **Peephole Optimization** option in the **PPC Processor** settings panel.
- For x86 code, this pragma is equivalent to the **Peephole Optimization** option in the **x86 Processor** settings panel.

poolsets

```
$pragma poolsets on | off | reset
```

Collects all set constants into a single data object. If this pragma is off, the compiler creates a unique data object.

For Mac OS Code Fragment Manager (CFM) code, turning this pragma on decreases the number of TOC entries in your program but increases your program's size, since it uses a less efficient method to store the string's address.

This pragma is especially useful if your program is large and has many set constants, or uses the CodeWarrior Profiler.

This pragma corresponds to the **Pool Sets** option in the Pascal Language preferences panel.

poolstrings

```
$pragma poolstrings on | off | reset
```

Collects all string literals into a single data object.

Setting this pragma to on does the following:

- The compiler collects all string constants into a single data object.

- Increases your program's size, since it uses a less efficient method to store the string's address.

For Mac OS Code Fragment Manager Code (CFM) code, setting this pragma to `off` means the compiler creates a unique data object and TOC entry for each string constant.

This option is especially useful if your program is large and has many string constants, or uses the CodeWarrior Profiler.

This pragma corresponds to the **Pool Strings** option in the Pascal Language preferences panel.

pop

```
$pragma pop
```

Restores (pops) the set of all the CodeWarrior Pascal compile-time settings that can be set using the IDE and/or pragmas. The pop pragma is used to restore the set of settings most recently saved with the push pragma (sets of settings are saved and restored in FIFO order).

The only limitation on the number of settings sets that can be pushed is available memory.

precompile_target

`$pragma precompile_target` *filename*

Specifies the file name of a precompiled file.

profile

```
$pragma profile on | off | reset
```

Generates extra instructions for profiling. This pragma is equivalent to the **Generate Profiler Calls** option in the **68K Processor** preference panel and the **Emit Profiler Calls** in the **PPC Processor** preference panel. To set up a project for the Profiler, turn this pragma on. This will cause CodeWarrior to insert special object code into your

application that collects timing information for the CodeWarrior Profiler. The default setting is off.

When you're ready to create the finished version of your application, ensure this pragma is turned off to eliminate the profiler code. You should also turn this pragma off when you use the CodeWarrior debugger, since the profiler code conflicts with the debugger.

propagation

```
$pragma propagation on | off | reset
```

Keeps track of variables known to be equivalent either to another variable or to a constant. If possible, the compiler replaces all uses of such a variable with its equivalent and eliminates the assignment to the variable, saving both execution time and code size. The default setting for this pragma is off.

This pragma is equivalent to the **Copy/Const Propagation** option of the **Global Optimizations** settings panel.

This pragma is available for 68K and x86 programming only.

ptrbasedobject

```
$pragma ptrbasedobject on | off | reset
```

Uses pointer based objects. If this pragma is off, the compiler uses handle based objects. This pragma must be set for the entire class hierarchy to be effective.

This pragma corresponds to the **Pointer Based Objects** option in the Pascal Language settings panel.

push

```
$pragma push
```

Saves (pushes) the set of all the CodeWarrior environment settings that can be set using the IDE and/or pragmas. The `push` pragma is used to save the current settings.

The only limitation on the number of settings sets that can be pushed is available memory.

rangecheck

```
$pragma rangecheck on | off | reset
```

Checks array indexes and variable assignments to make sure they are within their defined ranges. Using this pragma increases the size of your code, and decreases the execution speed.

This option is equivalent to the **Activate Range Checking** in the **Pascal Language Settings**.

readonlystrings

```
$pragma readonlystrings on | off | reset
```

Determines where the compiler stores string constants. If this pragma is turned off, the compiler stores string constants in the data section. If this pragma is on, the compiler stores string constants in the code section.

NOTE: Variables that are not initialized to the address of another object at run time are always placed in the code section.

This pragma corresponds to the **Make Strings ReadOnly** option in the **PPC Processor** preferences panel. The default setting for this pragma is off.

This pragma is available for PowerPC programming only.

regcoloring

```
$pragma regcoloring on | off | reset
```

Uses register coloring to do register assignment.

This pragma corresponds to the **Register Coloring** option on the **x86 CodeGen** preferences panel. The default setting of this pragma is off.

This pragma is available for x86 programming only.

relaxedptrcheck

```
$pragma relaxedptrcheck on | off | reset
```

Relaxes pointer compatibility rules by making pointer type variables compatible when they point to the same base type.

This pragma corresponds to the **Relax Pointer Compatibility** option in the **Pascal Language Settings** panel.

scheduling

```
$pragma scheduling 601 | 603 | 604 | on | off |
reset
```

Rearranges instructions so that the execution of one instruction doesn't hold up the execution of others, whenever possible.

You can choose the type of scheduling that works best for a specific PowerPC chip: 601, 603, 604, or off (no scheduling). Turning scheduling on turns instruction scheduling back on with the setting used (601, 603, 604) when instruction scheduling was last turned on.

WARNING! If you're debugging your code, ensure that this pragma is turned off. Because it rearranges the instructions produced from your code, the debugger will not be able to match the statements in your source code to the produced instructions with this pragma turned on.

This pragma is equivalent to the **Instruction Scheduling** option in the **PPC Processor** preferences panel. Its default setting is off.

This pragma is available for PowerPC programming only.

shortcircuit

```
$pragma shortcircuit on | off | reset
```

Evaluates logical operators are using short circuit evaluation (instead of full evaluation).

For more information on short circuit evaluation and logical operators, see your *Pascal Language Reference.*

sideeffects

```
$pragma sideeffects on | off | reset
```

Respects pointer aliases. If your program does not contain pointer alias, turn off this pragma to make your program smaller and faster. If your program does use pointer aliases, turn on this pragma to avoid incorrect code.

Listing 4.5 shows an example of a pointer alias.

Listing 4.5 Pointer alias example

```
var
  a, x : integer;
  p : ^integer;
begin
  p = @a; (* ^p is an alias for a *)
 (* ... *)
```

Remember that the compiler needs to load a variable into a register before performing operating on it. So, in the example in Listing 4.6, the compiler loads `a` into a register before the first addition. If `*p` is an alias for `a`, the compiler needs to load `a` into a register again before the second addition, since changing `*p` also changes `a`. If `*p` is not an alias for `a`, the compiler doesn't need to load `a` into a register again, since changing `*p` does not change `a`.

Listing 4.6 Example of a side effect with a pointer alias

```
x = a + 1;
*p = 0;      // Since *p is an alias for a, a is assigned 0.
y = a + 2;   // The register containing a must first be reloaded.
```

NOTE: The PowerPC compilers ignore this pragma and always assume that a program may contain pointer aliases.

smart_code

```
$pragma smart_code
```

This is simply a shortcut to setting the codesize pragma to smart, like this:

```
$pragma codesize smart
```

See "codesize" on page 68 for more information.

This pragma is available for 68K programming only.

strengthreduction

```
$pragma strengthreduction on | off | reset
```

Converts multiply operations inside loops into addition operations when possible. This optimization saves execution time, but may increase the size of your compiled program. The default setting for this pragma is off.

This pragma corresponds to the **Reduction in Strength** option of the **Global Optimizations** preferences panel.

This pragma is available for 68K and x86 programming only.

strsize

```
$pragma strsize on | off | reset
```

Copies strings according to their run-time length. When this pragma is off, the compiler copies strings according to their declared length.

This pragma corresponds to the **Copy Strings Using Length Byte** option in the Pascal Language preferences panel.

subexpression

```
$pragma subexpression on | off | reset
```

Detects cases in which the same expression is computed more than once, and eliminate the redundant computations. This can generally save both execution time and code size. The default setting for this pragma is off.

This pragma is equivalent to the **Common Sub-Expressions** option in the **Global Optimizations** preferences panel.

This pragma is available for 68K and x86 programming only.

sym

```
$pragma sym on | off | reset
```

Generates symbolic debugging information. This pragma is almost equivalent to the **Generate Sym File** option, which is always located in the Linker preference preferences panel for your project.

The difference is that **Generate Sym File** generates a SYM file containing all of the symbols in your project. You can generate a far more targeted SYM file by using the sym pragma like this in your source code:

```
$pragma sym on
 [function/routine declarations]
$pragma sym off
```

to toggle the generation of symbol information on and off for specific functions.

syminfo

```
$pragma syminfo on | off | reset
```

Generates symbolic information in SYM format for your project.

NOTE: (You must turn on both the syminfo pragma and the sym pragma to create a SYM file for debugging. To successfully generate a SYM file, turn the sym pragma on first, then turn the syminfo pragma on, and do not turn the sym pragma off.

This pragma is equivalent to the **SYM Format** option in the **x86 CodeGen** preferences panel.

This pragma is available for x86 programming only.

toc_data

```
$pragma toc_data on | off | reset
```

Stores static variables that are 4 bytes long or smaller directly in the TOC, instead of allocating space for them elsewhere and storing pointers to them in the TOC. You should turn this pragma on unless your code expects the TOC to contain pointers to data.

This pragma is equivalent to the **Store Static Data in TOC** option in the **PPC Processor** preferences panel. Its default setting is on.

This pragma is available for PowerPC programming only.

traceback

```
$pragma traceback on | off | reset
```

Generates an AIX-format traceback table for each function; these tables are placed in the executable code. Both the CodeWarrior and Apple debuggers can use traceback tables.

This option correspond to the **Emit Traceback Tables** option in the **PPC Processor** preferences panel. The default setting for this pragma is off.

trapnil

```
$pragma trapnil on | off | reset
```

Generates code to check for nil pointer dereferencing. Enabling this option increases code size and slows execution speed.

This pragma is equivalent to the **Activate Nil Checking** option in the **Pascal Language Settings** panel.

trapnoelse

```
$pragma trapnoelse on | off | reset
```

When this pragma is on, the compiler generates code to trap case expressions that do not match any case statement labels or `otherwise` clause. Enabling this option increases code size and slows execution speed.

This pragma is equivalent to the **Trap Unmatched Case** option in the **Pascal Language Settings** panel.

usespropagation

```
$pragma usespropagation on | off | reset
```

Extends the scope of unit objects. Suppose your program uses units Alpha and Beta, that Beta in turn uses Ceti, and that your program does not directly reference the Ceti unit in its USES clause. If you do not turn this pragma on, the routines in Ceti will not be available to your program. If you turn it on, they will be available to your program.

This pragma is equivalent to the **Uses Propagation** check box in the **Pascal Language** preferences panel.

warn_checkgoto

```
$pragma warn_checkgoto on | off | reset
```

Enables `goto` statement checking.

This pragma corresponds to the **Gotos & Labels** option in the **Pascal Warnings** preferences panel.

warn_foridx

```
$pragma warn_foridx on | off | reset
```

Issues a warning when a loop index is modified inside the `for`-loop body.

This pragma corresponds to the **Modified For-loop Indexes** option in the **Pascal Warnings** preferences panel.

warn_gotocase

```
$pragma warn_gotocase on | off | reset
```

Issues a warning when a `goto` statement is not properly referenced from another `case` statement.

This pragma corresponds to the **Branching Between Case Legs** option in the **Pascal Warnings** preferences panel.

warn_gotofor

```
$pragma warn_gotofor on | off | reset
```

Issues a warning when a `goto` label can be accessed from outside the `for` statement.

This pragma corresponds to the **Branching Into For** option in the **Pascal Warnings** preferences panel.

warn_gotoif

```
$pragma warn_gotoif on | off | reset
```

Issues a warning when a `goto` statement jumps to the `else` part of an `if... then... else` statement.

This pragma corresponds to the **Branching Between If & Else** option in the **Pascal Warnings** preferences panel.

warn_gotowith

```
$pragma warn_gotowith on | off | reset
```

Issues a warning when a `goto` label can be accessed from outside the `with` statement.

This pragma corresponds to the **Branching Into With** option in the **Pascal Warnings** preferences panel.

warn_illpragma

```
$pragma warn_illpragma on | off | reset
```

Issues warnings for illegal pragmas. An illegal pragma consists of a name not recognized as an authentic pragma after `$pragma.`

The default setting of this pragma is off.

warn_noreturn

```
$pragma warn_noreturn on | off | reset
```

Issues a warning for a function that doesn't return a value.

This pragma corresponds to the **Function Returns** option in the **Pascal Warnings** preferences panel.

warn_undefroutine

```
$pragma warn_undefroutine on | off | reset
```

Issues a warning for routines declared but not defined in the unit.

This pragma corresponds to the **Undefined Routines** option in the **Pascal Warnings** preferences panel.

warn_unusedarg

```
$pragma warn_unusedarg on | off | reset
```

Issues a warning when it encounters an argument you declare but do not use. This check helps you find misspelled argument names and arguments you have written out of your program.

This pragma corresponds to the **Unused Arguments** option in the **Pascal Warnings** project preferences panel.

warn_unusedvar

```
$pragma warn_unusedvar on | off | reset
```

Issues a warning when it encounters a variable you declare but do not use. This check helps you find misspelled variable names and variables you have written out of your program.

This pragma corresponds to the **Unused Variables** option in the **Pascal Warnings** project preferences panel.

Checking Options

The preprocessor function `option()` lets you test the setting of most pragmas and options that control the compiler. You typically modify these settings using various panels in the Target Settings dialog box and check them at compile time using the `option()` function.

The syntax for this preprocessor function is:

`option(`*option-name*`)`

If the specified option is on, `option ()` returns `true`; otherwise, it returns `false`.

This function is useful when you want one source file to contain code that uses different option settings. Listing 4.7 shows how to compile one series of lines if you're compiling for machines with the MC68881 floating-point unit and another series if you're compiling for machines without out.

Listing 4.7 Example of using option()

```
{$IFC option(code68881)}
  (* Code for 68K chip with FPU *)
{$ELSEC}
  (* Code for any 68K processor *)
{$ENDC}
```

Table 4.1 Arguments for option()

This argument…	Corresponds to…
`a6frames`	**Generate A6 Stack Frames** option in the 68K Linker settings panel and pragma a6frames.
`ANS_conformance`	**ANS Conformance** option in the **Pascal Language Settings** and pragma ANS_conformance.
`casesensitive`	**Case Sensitive Identifiers** option in the **Pascal Language Settings** and pragma casesensitive
`chkstrparam`	**Check String Parameter Sizes** option in the **Pascal Warnings Settings** and pragma chkstrparam
`cfm68k_codegen`	No option. Returns true if the compiler is generating CFM 68K-compatible object code, false otherwise.
`code68020`	**68020 Codegen** option in the 68K Processor settings panel and pragma code68020.

This argument…	Corresponds to…
`code68881`	**68881 Codegen** option in the **68K Processor** settings panel and pragma code68881.
`codelisting`	**Machine Code Listing** option in the **x86 CodeGen** settings panel and pragma codelisting
`codeviewinfo`	**CodeView Format** option in the **x86 CodeGen** settings panel and pragma codeviewinfo
`copyparam`	**Copy Value Parameter** option in the **Pascal Language Settings** and pragma copyparam.
`d0_pointers`	d0_pointers pragma.
`deadcode`	**Dead Code Elimination** option in the **Global Optimizations** settings panel and pragma deadcode
`deadstore`	**Dead Store Elimination** option in the **Global Optimizations** settings panel and pragma deadstore
`debuglisting`	pragma debuglisting
`disable_register` (PowerPC only)	disable_register pragma.
`dont_reuse_sets`	**Don't Reuse Sets** option in the **Pascal Language Settings** and pragma dont_reuse_sets.
`dont_reuse_strings`	**Don't Reuse Strings** option in the **Pascal Language Settings** and pragma dont_reuse_strings.
`expandtables`	**Expand Method Tables** option in the **Pascal Language Settings** and pragma expandtables.
`export`	pragma export
`external`	pragma external
`externalvar`	pragma externalvar
`far_code`	**Far Code** option in the **68K Processor** settings panel and pragma `far_code`
`fardata`	**Far Data** option in the **68K Processor** settings panel and pragma `fardata`

This argument…	Corresponds to…
`farmtables`	**Far Method Tables** option in the **68K Processor** settings panel and pragma `farmtables`
`farstrings`	**Far String Constants** option in the **68K Processor** settings panel and pragma `farstrings`.
`force_active`	pragma `force_active`
`force_public_data`	pragma `force_public_data`
`fourbyteints`	**4-Byte Ints** option in the **68K Processor** settings panel and pragma fourbyteints.
`fp_contract`	**Use FMADD & FMSUB** option in the **PPC Processor** settings panel and pragma `fp_contract`.
`global_optimizer`	**Global Optimization** option in the PPC Processor settings panel and pragma `global_optimizer`.
`IEEEdoubles`	**8-Byte Doubles** option in the 68K Processor settings panel and pragma IEEEdoubles.
`inlinedispatching`	pragma `inlinedispatching`
`import`	pragma `import`
`internal`	pragma `internal`
`intrinsic`	pragma `intrinsic`
`IOmodel`	pragma `IOmodel`
`lib_export`	pragma `lib_export`
`lifetime`	**Lifetime Analysis** option in the **Global Optimizations** settings panel and pragma lifetime
`logopt`	pragma `logopt`
`loopinvariant`	**Loop Invariants** option in the **Global Optimizations** settings panel and pragma lifetime
`linksym`	a read-only option that is true when the link SYM option in the linker settings panel is set

This argument…	Corresponds to…
`little_endian`	No option. It is 1 if you're compiling for a little endian target (such as x86) and 0 if you're compiling for a big endian target (such as Mac OS).
`macsbug`	**MacsBug Symbols** option in the **68K Linker** settings panel and pragma `macsbug`.
`makemap`	pragma `makemap`
`mpwc`	**MPW C Calling Conventions** option in the **68K Processor** settings panel and pragma `mpwc`.
`mpwc_relax`	**Relax Pointer Compatibility** option in the **Pascal Language Settings** and pragma `relaxedptrcheck`.
`noreturn`	pragma `noreturn`
`objcheck`	pragma `objcheck`
`objectPascal`	Whether the compiler is compiling this file as a Object Pascal file. Related to the **Activate Object Pascal** option in the **Pascal Language Settings** panel, the pragma objectPascal, and the macro `__objectPascal__`.
`oldstyle_symbols`	**MacsBug Symbols** option in the **68K Linker** settings panel and pragma `oldstyle_symbols`
`optimizeclasses`	**Optimize Class Hierarchy** option in the **Pascal Language Settings** and pragma `optimizeclasses`.
`optimize_for_size`	**Optimize Space** option in the **Global Optimizations** panel and pragma `optimize_for_size`
`optimize_for_space`	**Optimize Space** option in the **Global Optimizations** panel and pragma `optimize_for_space`
`optimize_for_speed`	**Optimize Speed** option in the **Global Optimizations** panel and pragma `optimize_for_speed`
`overflow`	**Activate Overflow Checking** in the **Debugging Aids** option in the **Pascal Language Settings** and pragma `overflow`

This argument…	Corresponds to…
`pcrelstrings`	**PC-Relative Strings** option in the **68K Processor** settings panel and pragma `pcrelstrings`.
`peephole`	**Peephole Optimization** option in the **PPC Processor** settings panel and pragma `peephole`.
`poolsets`	**Pool Sets** option in the **Pascal Language Settings** and pragma `poolsets`
`poolstrings`	**Pool Strings** option in the **Pascal Language Settings** and pragma `poolstrings`
`precompile`	Whether or not the file is being pre-compiled.
`preprocess`	Whether or not the file is being pre-processed.
`profile`	**Generate Profiler Calls** option in the 68K Processor settings panel, **Emit Profiler Calls** option in the PPC Processor settings panel, and pragma `profile`.
`propagation`	**Copy/Const Propagation** option in the **Global Optimizations** panel and pragma `propagation`
`ptrbasedobject`	**Pointer Based Objects** option in the **Pascal Language Settings** and pragma `ptrbasedobject`
`rangecheck`	**Activate Range Checking** in the **Debugging Aids** option in the **Pascal Language Settings** and pragma `rangecheck`
`readonlystrings`	**Make String Literals Readonly** option in the **PPC Processor** settings panel and pragma `readonlystrings`.
`codeviewinfo`	**CodeView Format** option in the **x86 CodeGen** settings panel and pragma codeviewinfo
`regcoloring`	**Register Coloring** option in the **x86 CodeGen** and pragma `regcoloring`.
`shortcircuit`	pragma `shortcircuit`
`sideeffects`	pragma `sideeffects`

This argument…	Corresponds to…
`strengthreduction`	**Reduction in Strength** option in the **Global Optimizations** panel and pragma `strengthreduction`
`strsize`	**Copy Strings Using Length Byte** option in the **Pascal Language Settings** and pragma `strsize`.
`subexpression`	**Common Sub-Expressions** option in the **Global Optimizations** panel and pragma `subexpression`
`sym`	**Generate SYM Files** option in the 68K and PPC Linker settings panels and pragma `sym`**.**
`syminfo`	**SYM Format** option in the **x86 CodeGen** and pragma `syminfo`**.**
`toc_data`	**Store Static Data in TOC** option in the PPC Processor settings panel and pragma `toc_data`
`traceback`	pragma `traceback`
`trapnil`	**Activate Nil Checking** in the **Debugging Aids** option in the **Pascal Language Settings** and pragma `trapnil`
`trapnoelse`	**Trap Unmatched Case** in the **Debugging Aids** option in the **Pascal Language Settings** and pragma `trapnoelse`
`usespropagation`	**Uses Propagation** option in the **Pascal Language Settings** and pragma `usespropagation`
`warn_checkgoto`	**Gotos & Labels** section of the **Pascal Language Settings** panel and pragma `warn_checkgoto`
`warn_foridx`	**Modified For-loop Indexes** option of the **Pascal Language Settings** panel and pragma `warn_foridx`
`warn_gotocase`	**Branching Between Case Legs** option of the **Pascal Language Settings** panel and pragma `warn_gotocase`
`warn_gotofor`	**Branching Into For** option of the **Pascal Language Settings** panel and pragma `warn_gotofor`
`warn_gotoif`	**Branching Between If & Else** option of the **Pascal Language Settings** panel and pragma warn_gotoif

This argument…	Corresponds to…
`warn_noreturn`	**Function Returns** option of the **Pascal Language Settings** panel and pragma `warn_noreturn`
`warn_undefroutine`	**Undefined Routines** option of the **Pascal Language Settings** panel and pragma `warn_undefroutine`
`warn_illpragma`	pragma `warn_illpragma`
`warn_unusedarg`	**Unused Arguments** option in the **Pascal Language Settings** panel and pragma warn_unusedarg.
`warn_unusedvar`	**Unused Variables** option in the **Pascal Language Settings** panel and pragma warn_unusedvar.

Predefined Symbols

Use predefined symbols to determine the kind of compiler and computer environment the source code is being compiled by. Predefined symbols do not choose the compiler or computer environment. Instead, they are used with the compilation control directives to control the compilation process to allow the programmer to take advantage of unique compiler and computer architecture features.

Another way to examine the compile-time environment is to use the option() preprocessor function. See "Checking Options" on page 98.

For more information on controlling compilation, see "Compiler Directives" on page 39.

The available predefined symbols are listed in Table 4.2.

Table 4.2 Predefined symbols in CodeWarrior Pascal

This macro...	is defined as...
`__A5__`	1 if the compiler generates 68K object code that is A5-relative, 0 if object code is A4-relative or if you are not compiling with a 68K compiler. In CodeWarrior C/C++, this macro is comparable to the `__A5__` predefined macro.
`CFM68K`	the same as `MAC68K`.
`__CFM68K__`	the same as `MAC68K`.
`__fourbyteints__`	1 if the compiler uses 4 bytes for the `int` data type as it generates object code for 68K, 0 if the compiler uses 2 bytes for the `int` data type or the compiler generates object code for some other platform. This macro corresponds to the **4-byte Ints** option in the 68K Processor settings panel. In CodeWarrior C/C++ this macro is comparable to the `__fourbyteints__` predefined macro.
`__IEEEdoubles__`	1 if the compiler uses 8 bytes for the `double` data type as it generates object code for the 68K, 0 if the compiler uses 10bytes for the `double` data type or the compiler generates object code for some other platform. This macro corresponds to the **4-byte Ints** option in the 68K Processor settings panel. In CodeWarrior C/C++ this macro is comparable to the `__IEEEdoubles__` predefined macro.
`INTEL`	1 if the compiler generates code for x86 and 0 for all other target platforms. In CodeWarrior C/C++ this macro is comparable to the `__INTEL__` predefined macro.

This macro...	is defined as...
`__INTEL__`	the same as `__INTEL__`
`MAC68K`	1 if the compiler generates object code for 68K, 0 if the compiler generates object code for some other platform. In CodeWarrior C/C++ this macro is comparable to the `__MC68K__` predefined macro.
`__MAC68K__`	the same as `MAC68K`.
`macintosh`	1 if the compiler's host platform is Mac OS computer, 0 if it is running on another platform. The compiler may be generating object code for any platform, including 68K, PowerPC, or Intel.
`__MC68020__`	1 if the compiler generates object code for the MC68020 or greater processor, 0 if object code is being generated for the MC68000 or the compiler generates object code for some other platform. This macro corresponds to the **68020 Codegen** option in the Processor settings panel. In CodeWarrior C/C++ this macro is comparable to the `__MC68020__` predefined macro.
`__MC68881__`	1 if the compiler generates object code for the MC68881 or greater math co-processor, 0 if object code is being generated to use a math library or the compiler generates object code for some other platform. This macro corresponds to the **68881 Codegen** option in the Processor settings panel. In CodeWarrior C/C++ this macro is comparable to the `__MC68881__` predefined macro.
`__MC68K__`	the same as `MAC68K`.

This macro...	is defined as...
`__MWBROWSER__`	defined as `1` if the IDE's browser is active or 0 if it is turned off. This macro corresponds to the **Activate Browser** option in the Extras settings panel. In CodeWarrior C/C++ this macro is `__MWBROWSER__`.
`MWERKS`	the version number of the CodeWarrior Pascal compiler, if you're using CodeWarrior CW7 or later. For example, in CodeWarrior Pascal version 1.9 __MWERKS__ would be `0x1900`. This macro is defined as 1 if the compiler was issued before CodeWarrior CW7. In CodeWarrior C/C++ this macro is comparable to `MWERKS`.
`__MWERKS__`	the same as `MWERKS`.
`__objectPascal__`	defined if you're compiling this file as a Object Pascal file, undefined if you're compiling this file as a regular Pascal file.
`POWERPC`	1 if the compiler generates object code for PowerPC processors, 0 if the compiler generates object code for some other platform. In CodeWarrior C/C++ this macro is `MWERKS`. In CodeWarrior C/C++ this macro is comparable to `__POWERPC__`.
`__powerpc`	same as `POWERPC`.
`__POWERPC__`	same as `POWERPC`.
`__profile__`	`1` if you turn on the **Generate Profiler Calls** option in the Processor settings panel or `0` if you turn that option off. In CodeWarrior C/C++ this macro is comparable to `__profile__`.

Index

Symbols

$D 42
$I 47
$IFC 47, 49
$pragma statements
    syntax of 63
$SETC 58
$U 58
.p 31
.pas 31
__option(), preprocessor function 99

Numerics

68000 processor
    Excluding code from executing on 52

A

a6frames 63
About
    command-line compilers 29
Activate Object Pascal 17
align 64
AND 58
ANS Conformance 17
ANS_conformance 65
ARRAY 37

B

BOOLEAN 36
Branching Between Case Legs 28
Branching Between If & Else 28
Branching Into For 27
Branching Into With 27

C

C
    calling routines from Pascal
        overview 32
        steps 33
    Data types
        matching with Pascal 36
C++
    calling routines from Pascal
        overview 34
        restrictions 35
        steps 34
    data types
        matching with Pascal 36
Case Sensitive Identifiers 17
Case sensitivity 57
casesensitive 65
Chapter overview 10
CHAR 36
Check String Parameter Sizes 26
chkstrparam 66
code68020 66
code68881 66
codesize 68
codeviewinfo 68
Compilation control directives 41, 42, 46, 50, 51, 52, 53, 55, 57, 58, 60
Compiler directives
    $ALIGN 41
    $D 42
    $ELSEC 44, 45
    $ENDC 45
    $EXPORT 46
    $I 47
    $IFC 47, 49
    $J 50
    $LIBEXPORT 50
    $MACRO 51
    $MAIN 51
    $MC68020 51
    $MC68881 52
    $NR 52
    $OV 53
    $PARAM 53
    $paramstr 53
    $PROF 55
    $PUSH 55
    $R 55
    $S 57
    $SC 58
    $SETC 58
    $U 58
    $undefined 59

$UNUSED 60
$W 60
$Z 60
Conditional statements 58
Copy Strings Using Length Byte 18
Copy Value Parameter 18
copyparam 69

D

d0_pointers 69
Data types
ARRAY 37
BOOLEAN 36
CHAR 36
DOUBLE 36
INTEGER 36
LONGINT 37
matching Pascal and C/C++ 36
REAL 36
RECORD 37
UNSIGNEDLONG 37
UNSIGNEDWORD 36, 37
deadcode 69
deadstore 69
disable_register 70
Don't Reuse Sets 19
Don't Reuse Strings 19
dont_reuse_sets 70
dont_reuse_strings 70
DOUBLE 36

E

Expand Method Tables 19
expandtables 71
export 71
EXTERNAL 33, 35
external 72
externalvar 72

F

far_code 72
fardata 72
farmtables 73
farstrings 73
File Mappings panel 31
file name extensions 31
Floating point unit
Code generation 52
force_active 74
fp_contract 75
Function Returns 24
example code 24

G

global_optimizer 75

I

IDE 31
IDE User Guide 31
IEEEdoubles 76
import 76
improving compiler performance 32
Inline Method Dispatching 19
Input file control 47, 58
INTEGER 36
IO Model 20
IOmodel 77

L

lib_export 78
lifetime 78
LONGINT 37
loopinvariant 78

M

MacsBug 42
macsbug 79
Manual conventions
Source code 12
Typographic 12
mark 79
Modified For-loop Indexes 24
example code 24

N

near_code 80
New Project command, IDE 31

noreturn 80

O

objcheck 80
objectPascal 81
oldstyle_symbols 81
optimization_level 82
Optimize Class Hierarchy 20
optimize_for_size 83
optimize_for_space 83
optimize_for_speed 84
optimizeclasses 82
`__option()`, preprocessor function 99
OR 58
Other documentation resources
  IDE User's Guide 11
  pointers to 11
  Quick Start Guide 11
Overflow 53
overflow 84

P

Parsing control directives 44, 45, 47, 49, 52, 53, 55, 58, 59, 60
Pascal
  calling C routines from 32
  calling C++ routines from 34
  Language settings overview 15
  matching data types with C/C++ 36
  Settings overview 15
Pascal Language settings
  Activate Object Pascal 17
  ANS Conformance 17
  Case Sensitive Identifiers 17
  Copy Strings Using Length Byte 18
  Copy Value Parameter 18
  Don't Reuse Sets 19
  Don't Reuse Strings 19
  Expand Method Tables 19
  Inline Method Dispatching 19
  IO Model 20
  Optimize Class Hierarchy 20
  Pointer Based Objects 20
  Pool Sets 20
  Pool Strings 21
  Prefix File 21
  Relax Pointer Compatibility 21
  Uses Propagation 22
Pascal language warnings
  branching between Case legs 28
  branching between If & Else 28
  branching into For 27
  branching into With 27
  check string parameter sizes 26
  function returns 24
  modified For-loop Indexes 24
  overview for gotos and labels 27
  undefined routines 26
  unused arguments 25
  unused variables 25
pcrelstrings 85
peephole 86
Pointer Based Objects 20
Pool Sets 20
Pool Strings 21
pool_sets 86
poolstrings 86
pop 87
Pragmas
  a6frames 63
  align 64
  ANS_conformance 65
  casesensitive 65
  chkstrparam 66
  code68020 66
  code68881 66
  codesize 68
  codeviewinfo 68
  copyparam 69
  d0_pointers 69
  deadcode 69
  deadstore 69
  disable_register 70
  dont_reuse_sets 70
  dont_reuse_strings 70
  expandtables 71
  export 71
  external 72
  externalvar 72
  far_code 72
  fardata 72

farmtables 73
farstrings 73
force_active 74
fp_contract 75
global_optimizer 75
IEEEdoubles 76
import 76
IOmodel 77
lib_export 78
lifetime 78
loopinvariant 78
macsbug 79
mark 79
near_code 80
noreturn 80
objcheck 80
objectPascal 81
oldstyle_symbols 81
optimization_level 82
optimize_for_size 83
optimize_for_space 83
optimize_for_speed 84
optimizeclasses 82
overflow 84
pcrelstrings 85
peephole 86
pool_sets 86
poolstrings 86
pop 87
profile 87
propagation 88
ptrbasedobject 88
push 88
rangecheck 89
readonlystrings 89
regcoloring 89
relaxedptrcheck 90
scheduling 90
shortcircuit 91
smart_code 92
strengthreduction 92
strsize 93
subexpression 93
sym 93
syminfo 94
toc_data 94
trapnil 95
trapnoelse 95
usespropagation 95
warn_checkgoto 96
warn_foridx 96
warn_gotocase 96
warn_gotofor 96
warn_gotoif 97
warn_gotowith 97
warn_illpragma 97
warn_noreturn 97
warn_undefroutine 98
warn_unusedarg 98
warn_unusedvar 98
precompiled units 32
Prefix File 21, 32
profile 87
propagation 88
ptrbasedobject 88
push 88

R

Range checking 55
rangecheck 89
readonlystrings 89
REAL 36
RECORD 37
regcoloring 89
Relax Pointer Compatibility 21
example code 21, 22
relaxedptrcheck 90
Resources 56

S

scheduling 90
Segmentation 56
shortcircuit 91
Short-circuit conditions 58
smart_code 92
speeding up the compiler 32
strengthreduction 92
strsize 93
subexpression 93
sym 93
syminfo 94

T

Targeting platforms
    Overview 11
toc_data 94
trapnil 95
trapnoelse 95

U

UNDEFINED 48, 49
Undefined Routines 26
Unit
    naming 59
UNSIGNEDLONG 37
UNSIGNEDWORD 36, 37
Unused Arguments 25
Unused Variables 25
    example code 25
Uses Propagation 22
usespropagation 95

V

Variables
    compile-time 58

W

warn_checkgoto 96
warn_foridx 96
warn_gotocase 96
warn_gotofor 96
warn_gotoif 97
warn_gotowith 97
warn_illpragma 97
warn_noreturn 97
warn_undefroutine 98
warn_unusedarg 98
warn_unusedvar 98

CodeWarrior

Pascal Compilers Reference

Credits

writing lead: Marc Paquette

other writers: Apple Computer Inc., Gavriel State, Patrick O'Connell

engineering: Marcel Achim, Nadine Bisson, Michael Cordery, Alain Duriez, Francois Plouffe

frontline warriors: Jim Trudeau, Ron Liechty, Ray Pierce, and CodeWarrior users everywhere

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting BeOS	How to use CodeWarrior to program for BeOS
Targeting Java VM	How to use CodeWarrior to program for the Java Virtual Machine
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows